目　录

第一章 图像及数字处理

1.1 引言

视觉是人类从大自然中获取信息的最主要的手段。据统计，在人类获取的信息中，视觉信息约占60%，听觉信息约占20%，其他的如味觉信息、触觉信息等加起来约占20%。由此可见视觉信息对人类的重要性，而图像正是人类获取视觉信息的主要途径。所谓“图”，就是物体透射或者反射光的分布；“像”是人的视觉系统接收图的信息而在大脑中形成的印象或认识。前者是客观存在的，而后者是人的感觉，图像应该是两者的结合。因此，在图像处理中不能仅仅把图像看成是二维平面上或者三维立体空间中具有明暗或色彩变化的光分布。

所谓图像处理，就是对图像信息进行加工以满足人的视觉心理或应用需求的行为。图像处理的手段有光学方法和电子学（数字）方法。前者已经有很长的发展历史，从简单的光学滤波到现在的激光全息技术，光学处理理论已经日趋完善，而且处理速度快，信息容量大，分辨率高，又很经济。但是光学处理图像精度不够高，稳定性差，操作不便。从20世纪60年代起，随着电子技术和计算机技术的不断提高和普及，数字图像处理进入高速发展时期。所谓数字图像处理就是利用数字计算机或者其他数字硬件，对从图像信息转换而得的电信号进行某些数学运算，以提高图像的实用性。例如从卫星图片中提取目标物的特征参数，三维立体断层图像的重建等等。数字图像处理技术处理精度比较高，而且还可以通过改进处理软件来优化处理效果。但是，由于数字图像处理的数据量非常庞大，因此处理速度相对较慢，这就限制了数字图像处理的发展。随着计算机技术的飞速发展，计算机的运算速度大大提高，目前1GHz以上的CPU已经开始推广应用，这将大大促进数字图像处理技术的发展。

1.2 数字图像处理概述

数字图像处理的英文名称是“Digital Image Processing”。通常所说的数字图像处理是指用计算机进行的处理，因此也称为计算机图像处理（Computer Image Processing）。总的来说，数字图像处理包括以下几项内容：

(1) 点运算

点运算主要是针对图像的像素进行加、减、乘、除等运算。图像的点运算可以有效地改变图像的直方图分布，这对提高图像的分辨率以及图像均衡都是非常有益的。

(2) 几何处理

几何处理主要包括图像的坐标转换，图像的移动、缩小、放大、旋转，多个图像的配准以及图像扭曲校正等。几何处理是最常见的图像处理手段，几乎任何图像处理软件都提供了

最基本的图像缩放功能。图像的扭曲校正功能可以将变形的图像进行几何校正，从而得出准确的图像。

(3) 图像增强

图像增强的作用主要是突出图像中重要的信息，同时减弱或者去除不需要的信息。常用方法有直方图增强和伪彩色增强等。

(4) 图像复原

图像复原的主要目的是去除干扰和模糊，从而恢复图像的本来面目。例如去噪声复原处理。

(5) 图像形态学处理

图像形态学是数学形态学的延伸，是一门独立的研究学科。利用图像形态学处理技术，可以实现图像的腐蚀、细化和分割等效果。

(6) 图像编码

图像编码研究属于信息论中信源编码的范畴，其主要宗旨是利用图像信号的统计特性及人类视觉特性对图像进行高效编码，从而达到压缩图像的目的。图像编码是数字图像处理中一个经典的研究范畴，有 60 多年的研究历史，目前已经制定了多种编码标准，如 H.261、JEPG 和 MEPG 等等。

(7) 图像重建

图像的重建起源于 CT 技术的发展，是一门新兴的数字图像处理技术，主要是利用采集的数据来重建出图像。图像重建的主要算法有代数法、迭代法、傅立叶反投影法和使用最广泛的卷积反投影法等。

(8) 模式识别

模式识别也是数字图像处理的一个新兴的研究方向，当今的模式识别方法通常有 3 种：统计识别法、句法结构模式识别法和模糊识别法。目前应用广泛的文字识别（OCR）技术就是应用模式识别技术开发出来的。

目前数字图像处理的应用越来越广泛，已经渗透到工业、医疗保健、航空航天、军事等各个领域，在国民经济中发挥越来越大的作用。

其中最典型的应用有：

(1) 遥感技术中的应用

遥感图像处理的用处已越来越大，并且其效率和分辨率也越来越高。它被广泛地应用于土地测绘、资源调查、气象监测、环境污染监督、农作物估产和军事侦察等领域。目前遥感技术已经比较成熟，但是还必须解决其数据量庞大、处理速度慢的缺点。

(2) 医学应用

图像处理在医学上有着广泛的应用。其中最突出的临床应用就是超声、核磁共振、γ 相机和 CT 等技术。在医学领域利用图像处理技术可以实现对疾病的直观诊断和无痛、安全方便的诊断和治疗，受到了广大患者的欢迎。

(3) 安全领域

利用图像处理的模式识别等技术，可以应用在监控、指纹档案管理等安全领域中。目前由清华大学工程物理系开发研制的大型集装箱检测系统，就是利用图像处理技术来实现全自动集装箱检测，从而加快了海关的工作效率，为打击走私立下汗马功劳。

(4) 工业生产

产品的无损检测也是图像处理技术的一项广泛的应用。

总之，图像处理技术的应用是相当广泛的，它在国家安全、经济发展、日常生活中充当着越来越重要的角色（有关图像处理的应用领域如表1－1所示），对国计民生有着不可忽略的作用。

表1－1　　图像处理的应用领域

学科	应用
物理、化学	结晶分析、谱分析等
生物、医学	细胞分析、染色体分类、X射线成像、CT等
环境保护	水质及大气污染调查等
地质	资源勘测、地图绘制、GIS等
农业、林业	农产物估产、植被分布调查等
渔业	鱼群分布调查等
气象	卫星云图分析等
通信	传真、电视、多媒体通信等
工业	工业探伤、机器人、产品质量监测等
军事	导弹导航、军事侦察等
法律	指纹识别等

第二章 Visual C++ 数字图像编程基础

本章将介绍 Windows 下 Visual C++数字图像编程的基础知识，它是后面章节编程学习的基础。主要介绍的内容有 Windows 位图的结构和调色板的概念；GDI 位图与设备无关位图的概念；如何构造自己的 DIB 函数库，以及如何用 Visual C++编程来实现 Windows 位图的读写。

2.1 图像和调色板

目前，Windows 系列操作系统（Windows 3.x、Windows 95/NT/2000）已经成为主流操作系统，它成功的一个重要原因是因为 Windows 有漂亮的人机交互界面和简便的操作。如果离开图形和图像，那么所有的 Windows 应用程序就会变得单调乏味。下面首先介绍一下图像究竟是怎样被显示出来的。

2.1.1 图像

普通的显示器屏幕是由许多的点构成的，这些点称为像素。显示时采用扫描的方式：电子枪每次从左到右扫描一行，为每个像素着色，然后再像这样从上到下扫描整个屏幕，利用人眼的视觉暂留效应就可以显示出一屏完整的图像。为了防止闪烁，每秒要重复上述扫描过程几十次。我们常说的屏幕分辨率为 1024×768，刷新频率为 85Hz，意思是每行扫描 1024 个像素，一共要扫描 768 行，每秒重复扫描屏幕 85 次。一般刷新频率大于 80Hz 时，人眼感受不到屏幕刷新而产生的闪烁，这种显示器被称为位映像设备。所谓位映像，就是指一个二维的像素矩阵，而位图就是采用位映像方法显示和存储的图像。图 2－1 是一幅普通的黑白位图，图 2－2 是被放大后的图，图中每个方格代表了一个像素，我们可以看到：整个图像就是由这样一些黑点和白点组成的。



图 2－1 人脸

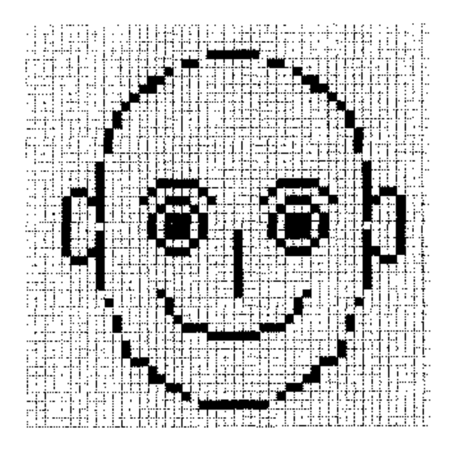


图 2－2 放大后的人脸位图

对于彩色图像，它的显示必须从三原色 RGB 概念说起。众所周知，自然界中的所有颜色都可以由红绿蓝（R、G、B）3 原色组合而成。有的颜色含有红色成分多一些，其他成分少一些。针对含有红色成分的多少，可以人为地分成 0 到 255 共 256 个等级，0 级表示不含红色成分，255 级表示含有 100%的红色成分。同样，绿色和蓝色也可以被分成 256 级。这样，根据红、绿、蓝各种不同的组合我们就能表示出 256×256×256（约 1 600 万）种颜色。

表 2－1 是常见的一些颜色的 RGB 组合值。

表 2－1　　常见颜色的 RGB 组合

颜色	红色成分	绿色成分	蓝色成分
黑色	0	0	0
白色	255	255	255
红色	255	0	0
绿色	0	255	0
蓝色	0	0	255
青色	0	255	255
紫色	255	0	255
黄色	255	255	0
灰色	128	128	128
橄榄色	128	128	0
深青色	0	128	128
银色	192	192	192

当一幅图中每个像素被赋予不同的 RGB 值时，就能呈现出五彩缤纷的颜色了，这就形成了彩色图像。

2.1.2 调色板

如果一幅图像的每个像素都用其 RGB 分量来表示，那么所有的图像文件都将变得非常庞大，实际上的做法不完全是这样的，可以先来看看一个简单的计算。

对一幅 200×200 的 16 色图像，它共有 40 000 个像素，如果每一个像素都用 *R*、*G*、*B* 三个分量表示，则一个像素需要 3 个字节（因为每个分量有 256 个级别，要用 8 位，即 1 个字节来表示，所以 3 个分量需要用 3 个字节）。这样保存整个图像要用 200×200×3，即 120 000 字节！但是如果采用下面的方法，就能省很多字节。

对于 16 色图像，图中最多只有 16 种颜色，如果采用一个颜色表：表中的每一行记录一种颜色的 *R*、*G*、*B* 值，这样当表示一个像素的颜色时，只需要指出该颜色是在第几行，即该颜色在表中的索引值便可以。例如，如果表的第 0 行为 255，0，0（红色），那么当某个像素为红色时，只需要标明 0 即可。通过颜色索引表来表示图像，来计算一下：16 种状态可以用 4 位（bit）表示，所以一个像素要用半个字节。整个图像要用 200×200×0.5，即 20 000 字节，再加上颜色表占用 3×16=48 字节，也不过 20 048 字节。这样一幅图像整个占用的字节数只是前面的 1/6！

其实这张 RGB 表，就是通常所说的调色板（Palette），它还有另外一种更确切的名称：

颜色查找表 LUT（Look Up Table）。在 Windows 位图中便用到了调色板技术，其实不仅仅是 Windows 位图，其他许多图像文件格式例如“.pcx”、“.tif”、“.gif”等都用到了调色板。所以很好地掌握调色板的概念是十分重要的。

还有一种情况，即真彩色图像（又叫做 24 位图像）的颜色种类高达 256×256×256=2^{24}=16 777 216 种，也就是包含上述提到的 *R*、*G*、*B* 颜色表示方法中所有的颜色。真彩色图像是说它具有显示所有颜色的能力，即最多可以包含所有的颜色。通常，在表示真彩色图时，每个像素直接用 *R*、*G*、*B* 这 3 个分量字节表示，而不采用调色板技术。原因很简单：如果使用调色板，表示一个像素颜色在调色板中的索引要用 24 位（因为共有 2^{24} 种颜色，即调色板有 2^{24} 行），这和直接用 *R*、*G*、*B* 这 3 个分量表示用的字节数一样，不但没有节省任何空间，还要加上一个 256×256×256×3 个字节的大调色板。所以真彩色图直接用 *R*、*G*、*B* 这 3 个分量表示。

2.1.3 色彩系统

前面介绍的 RGB 色彩系统是最常用的颜色系统，但在其他方面我们也会用到其他的色彩系统，常见的有：

1. RGB 和 CMY 色彩系统

CMY（Cyan、Magenta、Yellow）色彩系统也是一种常用的表示颜色的方式。计算机屏幕的显示通常用 RGB 色彩系统，它是通过颜色的相加来产生其他颜色，这种做法通常称为加色合成法（Additive Color Synthesis）。而在印刷工业上则通常用 CMY 色彩系统（一般所称的四色印刷 CMYK 则是加上黑色），它是通过颜色相减来产生其他颜色的，所以我们称这种方式为减色合成法（Subtractive Color Synthesis）。图 2—3 为 RGB 与 CMY 两个色彩系统的关系图：

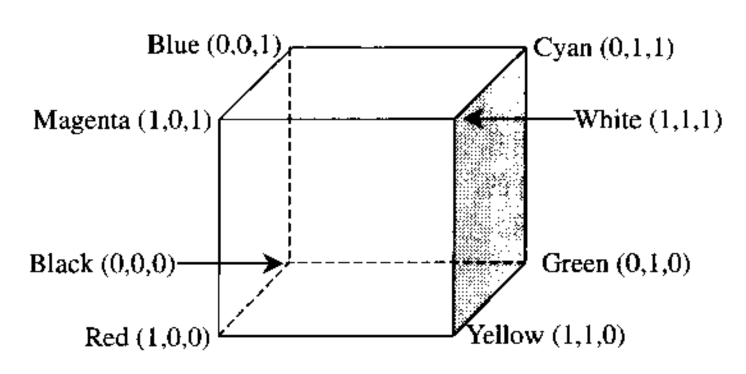


图 2—3 RGB 与 CMY 色彩系统关系图

2. YIQ 色彩系统

YIQ 色彩系统通常被北美的电视系统所采用（属于 NTSC 系统），这里 *Y* 不是指黄色，而是指颜色的明视度（Luminance），即亮度（Brightness）。其实 *Y* 就是图像的灰度值（Gray value），而 *I* 和 *Q* 则是指色调（Chrominance），即描述图像色彩及饱和度的属性。RGB 与 YIQ 之间的对应关系如下：

$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ 0.596 & -0.274 & -0.322 \\ 0.211 & -0.523 & 0.312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0.956 & 0.621 \\ 1 & -0.272 & -0.647 \\ 1 & -1.106 & -1.703 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

3. YUV 色彩系统

YUV 色彩系统被欧洲的电视系统所采用（属于 PAL 系统），其中 Y 和上面的 YIQ 色彩系统中的 Y 相同，都是指明视度。U 和 V 虽然也是指色调，但是和 I 与 Q 的表达方式不完全相同。RGB 与 YUV 之间的对应关系如下：

$$\begin{bmatrix} Y \\ U \\ V \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.148 & -0.289 & 0.437 \\ 0.615 & -0.515 & -0.100 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 0 & 1.140 \\ 1 & -0.395 & -0.581 \\ 1 & 2.032 & 0 \end{bmatrix} \begin{bmatrix} Y \\ U \\ V \end{bmatrix}$$

4. YcbCr 色彩系统

YcbCr 色彩系统也是一种常见的色彩系统，JPEG 采用的色彩系统正是该系统。它是从 YUV 色彩系统衍生出来的（因此通常还有人称 JPEG 采用的色彩系统是 YUV 系统，其实是错误的）。其中 Y 还是指明视度，而 Cb 和 Cr 则是将 U 和 V 做少量调整而得到的。RGB 色彩系统和 YcbCr 色彩系统之间的对应关系如下：

$$\begin{bmatrix} Y \\ Cb \\ Cr \\ 1 \end{bmatrix} = \begin{bmatrix} 0.2990 & 0.5870 & 0.1140 & 0 \\ -0.1687 & -0.3313 & 0.5000 & 128 \\ 0.5000 & -0.4187 & -0.0813 & 128 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 & 1.40200 & 0 \\ 1 & -0.34414 & -0.71414 \\ 1 & 1.77200 & 0 \end{bmatrix} \begin{bmatrix} Y \\ Cb-128 \\ Cr-128 \end{bmatrix}$$

2.1.4 灰度图

灰度图（Grayscale）是指只含亮度信息，不含色彩信息的图像，就像我们平时看到亮度由暗到明的黑白照片，变化是连续的。因此，要表示灰度图，就需要把亮度值进行量化。通常划分成 0 到 255 共 256 个级别，0 最暗（全黑），255 最亮（全白）。

BMP 格式的文件中并没有灰度图这个概念，但是我们可以很容易地用 BMP 文件来表示灰度图。方法是用 256 色的调色板，只不过这个调色板有点特殊，每一项的 RGB 值都是相同的，也就是说 RGB 值从（0，0，0），（1，1，1）一直到（255，255，255）。（0，0，0）是全黑色，（255，255，255）是全白色，中间的是灰色。这样，灰色度图就可以用 256 色图来表示了。对于 $R=G=B$ 的色彩，带入 YIQ 或 YUV 色彩系统转换公式中可以看到其颜色分量都是 0，即没有色彩信息。

灰度图使用比较方便。首先 RGB 的值都一样。其次，图像数据即调色板索引值，也就是实际的 RGB 的亮度值；另外因为是 256 色的调色板，所以图像数据中一个字节代表一个像素。如果是彩色的 256 色图，图像处理后有可能会产生不属于这 256 种颜色的新颜色，所以，图像处理一般采用灰度图。为了将重点放在算法上，今后给出的程序如不做特殊说明，都是针对 256 级灰度图的。

2.2 GDI 位图

前面介绍了一些关于图像颜色、调色板等的基本概念，下面将要介绍如何在 Visual C++ 中使用图像。首先介绍一下如何使用 GDI 位图。

GDI 是图形设备接口（Graphics Device Interface）的缩写。GDI 位图是一种 GDI 对象，在 Microsoft 基本类（MFC）库中用 CBitmap 类来表示。在 CBitmap 类对象中，包含一种和 Windows 的 GDI 模块有关的 Windows 数据结构，该数据结构是与设备相关的。应用程序可以得到 GDI 位图数据的一个备份，但是其中位的安排则完全依赖于显示设备。我们可以将 GDI 位图数据在同一台计算机内的不同应用程序间任意传递，但是由于其对设备的依赖性，在不同类型计算机间的传递是没有任何意义的。

图 2－4 是 CBitmap 类的继承关系图。CBitmap 类封装了 Windows GDI 位图，同时提供了一些操作位图的成员函数。在使用 CBitmap 对象时，首先要创建一个 CBitmap 对象，然后把它选进设备环境中，再调用其成员函数进行处理，在使用完毕后，把它从设备环境中选出并删除。

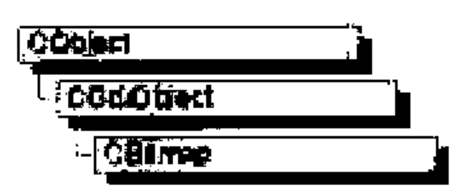


图 2－4　CBitmap 类的继承关系图

2.2.1　从资源中装入 GDI 位图

为了加载位图，可以使用 CBitmap 类的 LoadBitmap()成员函数。LoadBitmap 函数有两种调用方式：

```
BOOL LoadBitmap( LPCTSTR lpszResourceName );
BOOL LoadBitmap( UINT nIDResource );
```

一种是通过资源名称（由参数 lpszResourceName 指定）来加载指定的 GDI 位图；另外一种是通过资源 ID（由参数 nIDResource 指定）来加载指定的 GDI 位图。

假设工程中已经添加了一个 ID 为 IDB_BITMAP1 的位图资源，则用下面的简单的代码就可以显示该位图：

```
void CMyView::OnDraw(CDC* pDC)
{

    // CBitmap对象
    CBitmap bitmap;

    // CDC对象
    CDC dcMemory;

    // 加载资源
    bitmap.LoadBitmap(IDB_BITMAP1);

    // 创建内存设备环境
    dcMemory.CreateCompatibleDC(pDC);

    // 将位图选入内存设备环境中
    dcMemory.SelectObject(&bitmap);

    // 将内存设备环境复制到真正的设备环境中
    pDC->BitBlt(0,0,699,919,&dcMemory,0,0,SRCCOPY);

    // CDC析构函数退出前将删除dcMemory，位图选出

    // CBitmap析构函数删除位图

}
```

程序中 BitBlt()函数是将位图的像素从内存显示环境复制到显示器（或打印机）设备环境中。该函数十分有用，下面是它的函数原型：

```
BOOL BitBlt( int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, int ySrc, DWORD dwRop )
```

- 它的各个参数含义如下：

> x：指定绘制区域的左上角 x 坐标（逻辑单位）。
> y：指定绘制区域的左上角 y 坐标（逻辑单位）。
> nWidth：指定绘制区域的宽度。
> nHeight：指定绘制区域的高度。
> pSrcDC：指向要复制位图所在的 CDC 对象的指针。
> xSrc：指定原位图要绘制区域的左上角 x 坐标（逻辑单位）。
> ySrc：指定原位图要绘制区域的左上角 y 坐标（逻辑单位）。
> dwRop：指定绘制方式，其取值如表 2-2 所示。

表 2－2　　dwRop 参数的取值表

参数	含义
BLACKNESS	将所有输出变成黑色
DSTINVERT	反转目标位图
MERGECOPY	合并模式和原位图。
MERGEPAINT	用或（or）运算合并反转的原位图和目标位图。
NOTSRCCOPY	将反转的原位图复制到目标。
NOTSRCERASE	用或（or）运算合并原位图和目标位图，然后反转。
PATCOPY	将模式复制到目标位图。
PATINVERT	用异或（xor）运算合并目标位图与模式。
PATPAINT	用或（or）运算合并反转的原位图与模式。然后用或（or）运算合并上述结果与目标位图。
SRCAND	用与（and）运算合并目标像素与原位图。
SRCCOPY	将原位图复制到目标位图。
SRCERASE	反转目标位图并用与（and）运算合并所得结果与原位图。
SRCINVERT	用异或（xor）运算合并目标像素和原位图。
SRCPAINT	用或（or）运算合并目标像素和原位图。
WHITENESS	将所有输出变成白色

程序运行的结果如图 2－5 所示。



图 2－5　从资源中装入 GDI 位图示例

> 上面的代码产生的显示效果不错，打印预览也能看见图像，但是真正打印时就会发现什么也没打出来。这是因为代码是专门为了显示而构造的，无法将它选进与打印机兼容的内存设备环境中。如果打印位图，可以使用后面介绍的 DIB。

2.2.2 伸缩位图

有时，我们想对位图进行放大或缩小的操作，这时就可以使用 StretchBlt()函数来显示位图。下面是该函数原型：

```
BOOL StretchBlt( int x, int y, int nWidth, int nHeight, CDC* pSrcDC, int xSrc, ySrc, int nSrcWidth,
int nSrcHeight, DWORD dwRop )
```

该函数和 BitBlt()基本上一致，只是多了两个参数 nSrcWidth 和 nSrcHeight，用来指定要复制原图像的宽度和高度。

下面我们更改 2.2.1 节的代码，调用该函数显示出两个缩小的图像：

```
void CMyView::OnDraw(CDC* pDC)
{

    // CBitmap对象
    CBitmap bitmap;

    // CDC对象
    CDC dcMemory;

    // 加载资源
    bitmap.LoadBitmap(IDB_BITMAP1);

    // 创建内存设备环境
    dcMemory.CreateCompatibleDC(pDC);

    // 将位图选入内存设备环境中
    dcMemory.SelectObject(&bitmap);

    // 将内存设备环境复制到真正的设备环境中
    pDC->BitBlt(0,0,699,919,&dcMemory,0,0,SRCCOPY);

    // 缩小一半显示
    pDC->StretchBlt(699,0,699/2,919/2,&dcMemory,0,0,699,919,SRCCOPY);

    pDC->StretchBlt(699,919/2,699/2,919/2,&dcMemory,0,0,699,919,SRCCOPY);

    // CDC析构函数退出前将删除dcMemory，位图选出

    // CBitmap析构函数删除位图

}
```

这时再运行该程序，出现的结果如图 2－6 所示。

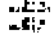




图 2－6　缩放位图示例

2.3　设备无关位图（DIB）

DIB 是 Device-Independent Bitmap（设备无关位图）的缩写，它自带颜色信息，因此调色板管理非常容易。DIB 也使打印时的灰度阴影的控制更加容易。任何运行 Windows 的计算机都可以处理 DIB，它通常以 BMP 文件的形式被保存在磁盘中或者作为资源保存在 EXE 文件和 DLL 文件中。

2.3.1　BMP 文件中 DIB 的结构

DIB 是标准的 Windows 位图格式，BMP 文件中包含了一个 DIB。一个 BMP 文件大体上分成如下 4 个部分，如图 2－7 所示。

BITMAPFILEHEADER	bfType = "BM"
位图文件头	bfSize
（只用于BMP文件）	bfReserved1
	bfReserved2
	bfOffBits
BITMAPINFOHEADER	biSize
位图信息头	biWidth
	biHeight
	biPlanes
	biBitCount
	biCompression
	biSizeImage
	biXPelsPerMeter
	biYPelsPerMeter
	biClrUsed
	biClrImportant
Palette	单色DIB有2个表项
调色板	16色DIB有16个表项或更少
	256色DIB有256个表项或更少
	真彩色DIB没有调色板
	每个表项长度为4字节（四字节）
DIB Pixels	像素按照每行每列的顺序排列
DIB图像数据	每一行的字节数必须是4的整倍数

图 2—7 BMP 文件结构示意图

第一部分为位图文件头 BITMAPFILEHEADER，它是一个结构，其定义如下：

```
typedef struct tagBITMAPFILEHEADER {
        WORD      bfType;
        DWORD     bfSize;
        WORD      bfReserved1;
        WORD      bfReserved2;
        DWORD     bfOffBits;
} BITMAPFILEHEADER, FAR *LPBITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

该结构的长度是固定的，为 14 个字节（WORD 为无符号 16 位整数，DWORD 为无符号 32 位整数），各个域的说明如下：

- bfType：指定文件类型，必须是 0x424D，即字符串“BM”，也就是说所有“.bmp”文件的头两个字节都是“BM”。
- bfSize：指定文件大小，包括这 14 个字节。
- bfReserved1，bfReserved2：为保留字，不用考虑。
- bfOffBits：为从文件头到实际的位图数据的偏移字节数，即图 2-7 中前三个部分的长度之和。

第二部分为位图信息头 BITMAPINFOHEADER，它也是一个结构，其定义如下：

```
typedef struct tagBITMAPINFOHEADER{
        DWORD      biSize;
        LONG       biWidth;
        LONG       biHeight;
        WORD       biPlanes;
        WORD       biBitCount;
        DWORD      biCompression;
        DWORD      biSizeImage;
        LONG       biXPelsPerMeter;
        LONG       biYPelsPerMeter;
        DWORD      biClrUsed;
        DWORD      biClrImportant;
} BITMAPINFOHEADER, FAR *LPBITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

该结构的长度也是固定的，为 40 个字节（WORD 为无符号 16 位整数，DWORD 为无符号 32 位整数，LONG 为 32 位整数）。各个域的说明如下：

- biSize：指定这个结构的长度，为 40 字节。
- biWidth：指定图像的宽度，单位是像素。
- biHeight：指定图像的高度，单位是像素。
- biPlanes：必须是 1，不用考虑。
- biBitCount：指定表示颜色时要用到的位数，常用的值为 1（黑白二色图）、4（16 色图）、8（256 色）、24（真彩色图），新的“.bmp”格式支持 32 位色，这里就不做讨论了。
- biCompression：指定位图是否压缩，有效的值为 BI_RGB，BI_RLE8，BI_RLE4，BI_BITFIELDS（都是一些 Windows 定义好的常量）。要说明的是，Windows 位图可以采用 RLE4 和 RLE8 的压缩格式，但用得不多。我们今后所讨论的只有第一种不压缩的情况，即 biCompression 为 BI_RGB 的情况。
- biSizeImage：指定实际的位图数据占用的字节数，其实也可以从以下的公式中计算出来：

  biSizeImage = biWidth' × biHeight

  要注意的是：上述公式中的 biWidth'必须是 4 的整倍数（所以不是 biWidth，而是 biWidth'，表示大于或等于 biWidth 的离 4 最近的整倍数。举个例子，如果 biWidth=240，则 biWidth'=240；如果 biWidth=241，biWidth'=244）。如果 biCompression 为 BI_RGB，则该项可能为零。
- biXPelsPerMeter：指定目标设备的水平分辨率，单位是每米的像素个数。
- biYPelsPerMeter：指定目标设备的垂直分辨率，单位是每米的像素个数。
- biClrUsed：指定本图像实际用到的颜色数，如果该值为零，则用到的颜色数为 2 的 biBitCount 次幂。
- biClrImportant：指定本图像中重要的颜色数，如果该值为零，则认为所有的颜色都是重要的。

第三部分为调色板（Palette）。有些位图需要调色板，有些位图，如真彩色图，不需要调色板，它们的 BITMAPINFOHEADER 后面直接是位图数据。

调色板实际上是一个数组，共有 biClrUsed 个元素（如果该值为零，则有 2 的 biBitCount 次幂个元素）。数组中每个元素的类型是一个 RGBQUAD 结构，占 4 个字节，其定义如下：

```
typedef struct tagRGBQUAD {
        BYTE    rgbBlue;
        BYTE    rgbGreen;
        BYTE    rgbRed;
        BYTE    rgbReserved;
} RGBQUAD;
```

其中：

- rgbBlue：该颜色的蓝色分量。
- rgbGreen：该颜色的绿色分量。
- rgbRed：该颜色的红色分量。
- rgbReserved：保留值。

第四部分就是实际的图像数据。对于用到调色板的位图，图像数据就是该像素颜色在调色板中的索引值，对于真彩色图，图像数据就是实际的 R、G、B 值。下面就 2 色、16 色、256 色位图和真彩色位图分别介绍。

- 对于 2 色位图，用 1 位就可以表示该像素的颜色（一般 0 表示黑，1 表示白），所以一个字节可以表示 8 个像素。
- 对于 16 色位图，用 4 位可以表示一个像素的颜色，所以 1 个字节可以表示 2 个像素。
- 对于 256 色位图，1 个字节刚好可以表示 1 个像素。
- 对于真彩色图，3 个字节才能表示 1 个像素。

要注意两点：

- **1. 每一行的字节数必须是 4 的整倍数，如果不是，则需要补齐。这在前面介绍 biSizeImage 时已经提到了。**
- **2. 一般来说，BMP 文件的数据是从下到上、从左到右的。即从文件中最先读到的是图像最下面一行的左边第一个像素，然后是左边第二个像素…接下来是倒数第二行左边第一个像素，左边第二个像素…依次类推，最后得到的是最上面一行的最右一个像素。**

2.3.2 DIB 访问函数

Windows 支持一些重要的 DIB 访问函数，但是这些函数都还没有被封装到 MFC 中。下面是这些 DIB 访问函数的简要介绍。

1. SetDIBitsToDevice

该函数可以直接在显示器或打印机上显示 DIB。在显示时不进行缩放处理，即位图的每一像素对应于一个显示器的显示像素或一个打印机的打印点。正是这个特点限制了该函数的使用，使它不能像函数 BitBlt()一样，因为后者使用的是逻辑坐标。

下面是该函数的原型：

```
int SetDIBitsToDevice( HDC hdc, int XDest, int YDest, DWORD dwWidth, DWORD dwHeight, int XSrc, int YSrc, UINT uStartScan, UINT cScanLines, CONST VOID *lpvBits, CONST BITMAPINFO *lpbmi, UINT fuColorUse )
```

其中各个参数的含义如下：

- hdc：设备上下文句柄。
- XDest：指定绘制区域的左上角 x 坐标（逻辑单位）。
- YDest：指定绘制区域的左上角 y 坐标（逻辑单位）。
- dwWidth：指定 DIB 的宽度（逻辑单位）。
- dwHeight：指定 DIB 的高度（逻辑单位）。
- XSrc：指定原位图要绘制区域的左上角 x 坐标（逻辑单位）。
- YSrc：指定原位图要绘制区域的左上角 y 坐标（逻辑单位）。
- uStartScan：指定 DIB 扫描的起始行。
- cScanLines：指定 DIB 扫描的行数（即 DIB 高度）。
- lpvBits：指向 DIB 数据图像的指针。
- lpbmi：指向 BITMAPINFO 结构的指针。
- fuColorUse：指定 BITMAPINFO 结构中的 bmiColors 包含真实的 RGB 值还是调色板中的索引值。它的取值可能是：
  DIB_PAL_COLORS：调色板中包含的是当前逻辑调色板的索引值。
  DIB_RGB_COLORS：调色板中包含的是真正的 RGB 数值。

该函数如果调用成功，返回绘制的行数；如果失败，则返回 0。

2. StretchDIBits

该函数类似于 StretchBlt，利用它可以缩放显示 DIB 于显示器和打印机上。它的函数原型如下：

```
int StretchDIBits( HDC hdc, int XDest, int YDest, int nDestWidth, int nDestHeight, int XSrc, int YSrc, int nSrcWidth, int nSrcHeight, CONST VOID *lpBits, CONST BITMAPINFO *lpBitsInfo, UINT iUsage, DWORD dwRo)
```

其中各个参数的含义如下：

- hdc：设备上下文句柄。
- XDest：指定绘制区域的左上角 x 坐标（逻辑单位）。
- YDest：指定绘制区域的左上角 y 坐标（逻辑单位）。
- nDestWidth：指定 DIB 的宽度（逻辑单位）。
- nDestHeight：指定 DIB 的高度（逻辑单位）。
- XSrc：指定原位图要绘制区域的左上角 x 坐标（逻辑单位）。
- YSrc：指定原位图要绘制区域的左上角 y 坐标（逻辑单位）。
- nSrcWidth：指定要复制原图像矩形区域的宽度（逻辑单位）。
- nSrcHeight：指定要复制原图像矩形区域的高度（逻辑单位）。

- lpvBits：指向 DIB 数据图像的指针。
- lpBitsInfo：指向 BITMAPINFO 结构的指针。
- iUsage：指定 BITMAPINFO 结构中的 bmiColors 包含真实的 RGB 值还是调色板中的索引值。它的取值可能是：
  DIB_PAL_COLORS：调色板中包含的是当前逻辑调色板的索引值。
  DIB_RGB_COLORS：调色板中包含的是真正的 RGB 数值。
- dwRo：指定绘制方式，其取值如表 2－2 所示。

该函数如果调用成功，返回绘制的行数；如果失败，则返回 GDI_ERROR。

3. GetDIBits

该函数利用申请到的内存，由 GDI 位图来构造 DIB。通过该函数，可以对 DIB 的格式进行控制，可以指定每个像素颜色的位数，而且可以指定是否进行压缩。如果采用了压缩格式，则必须调用该函数 2 次，一次用于计算所需要的内存，另外一次用于产生 DIB 数据。该函数的原型如下：

```
int GetDIBits( HDC hdc, HBITMAP hbmp, UINT uStartScan, UINT cScanLines, LPVOID lpvBits,
LPBITMAPINFO lpbi, UINT uUsage)
```

其中各个参数的含义如下：

- hdc：设备上下文句柄。
- hbmp：原位图的句柄。
- uStartScan：指定 DIB 扫描的起始行。
- cScanLines：指定 DIB 扫描的行数（即 DIB 高度）。
- lpvBits：指向 DIB 数据图像的指针。
- lpbi：指向 BITMAPINFO 结构的指针。
- uUsage：指定 BITMAPINFO 结构中的 bmiColors 包含真实的 RGB 值还是调色板中的索引值。它的取值可能是：
  DIB_PAL_COLORS：调色板中包含的是当前逻辑调色板的索引值。
  DIB_RGB_COLORS：调色板中包含的是真正的 RGB 数值。

该函数如果调用成功并且 lpvBits 非空，返回绘制的行数；如果失败，则返回 0。

4. CreateDIBitmap

利用该函数可以从 DIB 出发来创建 GDI 位图。该函数的原型如下：

```
HBITMAP CreateDIBitmap( HDC hdc, CONST BITMAPINFOHEADER *lpbmih, DWORD fdwInit,
CONST VOID *lpbInit, CONST BITMAPINFO *lpbmi, UINT fuUsage)
```

其中各个参数的含义如下：

- hdc：设备上下文句柄。
- lpbmih：指向位图信息头结构的指针。位图信息头结构可能为：
  BITMAPINFOHEADER（Windows NT 3.51 and earlier）
  BITMAPV4HEADER（Windows NT 4.0 and Windows 95）

BITMAPV5HEADER（Windows NT 5.0 and Windows 98）

- fdwInit：该参数指定是否按照特定的格式（由参数 lpbInit 和 lpvBits 来指定）来初始化位图。如果其取值为 CBM_INIT，则按照特定的格式初始化位图；如果该参数取值为 0，则不初始化位图。
- lpbInit：指向初始化位图的数据指针。
- lpbmi：指向初始化位图的 BITMAPINFO 结构的指针。
- fuUsage：指定 BITMAPINFO 结构中的 bmiColors 包含真实的 RGB 值还是调色板中的索引值。它的取值可能是：
  DIB_PAL_COLORS：调色板中包含的是当前逻辑调色板的索引值。
  DIB_RGB_COLORS：调色板中包含的是真正的 RGB 数值。

该函数如果调用成功，返回创建位图的句柄；如果失败，则返回 NULL。

5. CreateDIBSection

CreateDIBSection 函数能创建一种特殊的 DIB，称为 DIB 项（DIB Section），然后返回一个 GDI 位图句柄。它提供了 DIB 和 GDI 位图的最好的特性。这样我们可以直接访问 DIB 的内存，可以利用位图句柄和内存设备环境，我们甚至还可以在 DIB 中调用 GDI 函数来绘图。该函数的原型定义如下：

```
HBITMAP CreateDIBSection( HDC hdc, CONST BITMAPINFO *pbmi, UINT iUsage, VOID
*ppvBits, HANDLE hSection, DWORD dwOffset)
```

其中各个参数的含义如下：

- hdc：设备上下文句柄。
- pbmi：指向 BITMAPINFO 结构的指针。
- iUsage：指定 BITMAPINFO 结构中的 bmiColors 包含真实的 RGB 值还是调色板中的索引值。它的取值可能是：
  DIB_PAL_COLORS：调色板中包含的是当前逻辑调色板的索引值。
  DIB_RGB_COLORS：调色板中包含的是真正的 RGB 数值。
- ppvBits：指向创建的 DIB 图像数据的指针。
- hSection：指定创建 DIB 的文件映像对象的句柄，该参数可以为 NULL。
- dwOffset：指定由参数 hSection 指向的文件映像对象的偏移量，如果 hSection 为 NULL，则忽略该参数。

该函数如果调用成功，返回创建位图的句柄，并且参数 ppvBits 指向创建的 DIB 图像数据；如果失败，则返回 NULL，并且参数 ppvBits 也被赋值为 NULL。

6. LoadImage 函数

在 Windows 95 和 Windows NT 4.0 以及其后的所有 Windows 版本中，都提供了 LoadImage 函数。该函数可以直接从磁盘文件中读入一个位图，并返回一个 DIB 句柄。该函数的原型定义如下：

```
HANDLE LoadImage( HINSTANCE hinst, LPCTSTR lpszName, UINT uType, int cxDesired, int
```

cyDesired, UINT fuLoad)

其中各个参数的含义如下：

- hinst：包含要加载图像的实例。
- lpszName：要加载的图像的文件或资源名称。
- uType：要加载的图像类型。它的取值可能是：
  IMAGE_BITMAP：位图。
  IMAGE_CURSOR：光标。
  IMAGE_ICON：图标。
- cxDesired：指定要加载的光标和图标的像素宽度。
- cyDesired：指定要加载的光标和图标的像素高度。
- fuLoad：加载选项，它的取值是下列值的组合：
  LR_DEFAULTCOLOR：默认值，没有特殊的含义，表明不对图像颜色进行处理。
  LR_CREATEDIBSECTION：当参数 uType 取值为 IMAGE_BITMAP，该参数指定创建一个 DIB 项。
  LR_DEFAULTSIZE：指明使用图像默认大小。
  LR_LOADFROMFILE：指明是从由参数 lpszName 指定的文件中加载图像。如果不指明，默认是从由参数 lpszName 指定的资源中加载图像。
  LR_LOADMAP3DCOLORS：如果指定该选项，函数会自动把颜色 RGB（128、128、128）替换成 COLOR_3DSHADOW；把 RGB（192、192、192）替换成 COLOR_3DFACE；把颜色 RGB（223，223，223）替换成 COLOR_3DLIGHT。
  LR_MONOCHROME：把图像转换成黑白图像。
  LR_SHARED：如果图像被打开多次，则共享该图像句柄。
  LR_VGACOLOR：使用 VGA 颜色。

该函数如果调用成功，返回读取位图的句柄；如果失败，则返回 NULL。

7. DrawDibDraw 函数

Windows 提供了窗口视频（VFW，Video for Windows）组件，Visual C++支持该组件。VFW 中的 DrawDibDraw 函数是一个可以替代 StretchDIBits 的函数。它的最主要的优点是可以使用抖动颜色（Dithered Color），并且提高显示 DIB 的速度，缺点是必须将 VFM 代码连接到进程中。

该函数的原型定义如下：

```
BOOL DrawDibDraw( HDRAWDIB hdd, HDC hdc, int xDst, int yDst, int dxDst, int dyDst,
LPBITMAPINFOHEADER lpbi, LPVOID lpBits, int xSrc, int ySrc, int dxSrc, int dySrc, UINT wFlags)
```

其中各个参数的含义如下：

- hdd：DrawDib DC 的句柄。可以通过调用函数::DrawDibOpen()返回。
- hdc：设备上下文句柄。
- xDst：指定绘制区域的左上角 x 坐标。
- yDst：指定绘制区域的左上角 y 坐标。

- dxDst：指定 DIB 的宽度。
- dyDst：指定 DIB 的高度。
- ipbi：指向 BITMAPINFOHEADER 结构的指针。
- lpBits：指向 DIB 数据图像的指针。
- xSrc：指定原位图要绘制区域的左上角 *x* 坐标（像素）。
- ySrc：指定原位图要绘制区域的左上角 *y* 坐标（像素）。
- dxSrc：指定要复制原图像矩形区域的宽度（像素）。
- dySrc：指定要复制原图像矩形区域的高度（像素）。
- wFlags：绘制方式。

该函数如果调用成功，返回 TRUE；如果失败，则返回 FALSE。

2.3.3 构造自己的 DIB 函数库

虽然 MFC 没有封装 DIB，但是在程序中使用 DIB 还是十分方便的。在本小节中，将要构造我们自己的 DIB 函数库，这样在今后使用 DIB 时，只要自己调用函数库就可以了。下面是 DIB 函数库的头文件“DIBAPI.H”的内容。

```
// dibapi.h

#ifndef _INC_DIBAPI
#define _INC_DIBAPI

// DIB句柄
DECLARE_HANDLE(HDIB);

// DIB常量
#define PALVERSION    0x300

/* DIB宏 */

// 判断是否是Win 3.0的DIB
#define IS_WIN30_DIB(lpbi)   ((*(LPDWORD)(lpbi)) == sizeof(BITMAPINFOHEADER))

// 计算矩形区域的宽度
#define RECTWIDTH(lpRect)     ((lpRect)->right - (lpRect)->left)

// 计算矩形区域的高度
#define RECTHEIGHT(lpRect)    ((lpRect)->bottom - (lpRect)->top)

// 在计算图像大小时，采用公式：biSizeImage = biWidth' × biHeight。
// 是biWidth'，而不是biWidth，这里的biWidth'必须是4的整倍数，表示
// 大于或等于biWidth的，离4最近的整倍数。WIDTHBYTES就是用来计算
// biWidth'
#define WIDTHBYTES(bits)    (((bits) + 31) / 32 * 4)

// 函数原型
BOOL      WINAPI   PaintDIB (HDC, LPRECT, HDIB, LPRECT, CPalette* pPal);
```

```
BOOL      WINAPI  CreateDIBPalette(HDIB hDIB, CPalette* cPal);
LPSTR     WINAPI  FindDIBBits (LPSTR lpbi);
DWORD     WINAPI  DIBWidth (LPSTR lpDIB);
DWORD     WINAPI  DIBHeight (LPSTR lpDIB);
WORD      WINAPI  PaletteSize (LPSTR lpbi);
WORD      WINAPI  DIBNumColors (LPSTR lpbi);
HGLOBAL   WINAPI  CopyHandle (HGLOBAL h);

BOOL      WINAPI  SaveDIB (HDIB hDib, CFile& file);
HDIB      WINAPI  ReadDIBFile(CFile& file);

#endif //!_INC_DIBAPI
```

下面就是这些函数的实现源代码。

```
//************************************************************************
//  文件名：dibapi.cpp
//
//  DIB(Independent Bitmap) API函数库：
//
//  PaintDIB()          - 绘制DIB对象
//  CreateDIBPalette()  - 创建DIB对象调色板
//  FindDIBBits()       - 返回DIB图像像素起始位置
//  DIBWidth()          - 返回DIB宽度
//  DIBHeight()         - 返回DIB高度
//  PaletteSize()       - 返回DIB调色板大小
//  DIBNumColors()      - 计算DIB调色板颜色数目
//  CopyHandle()        - 拷贝内存块
//
//  SaveDIB()           - 将DIB保存到指定文件中
//  ReadDIBFile()       - 重指定文件中读取DIB对象
//
//************************************************************************


#include "stdafx.h"
#include "dibapi.h"
#include <io.h>
#include <errno.h>

#include <math.h>
#include <direct.h>

/*
 * Dib文件头标志（字符串"BM"，写DIB时用到该常数）
 */
#define DIB_HEADER_MARKER   ((WORD) ('M' << 8) | 'B')

/*************************************************************************
 *
 * 函数名称：
 *   PaintDIB()
```

```
 *
 * 参数:
 *    HDC hDC              - 输出设备DC
 *    LPRECT lpDCRect      - 绘制矩形区域
 *    HDIB hDIB            - 指向DIB对象的指针
 *    LPRECT lpDIBRect     - 要输出的DIB区域
 *    CPalette* pPal       - 指向DIB对象调色板的指针
 *
 * 返回值:
 *    BOOL                 - 绘制成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *    该函数主要用来绘制DIB对象。其中调用了StretchDIBits()或者
 * SetDIBitsToDevice()来绘制DIB对象。输出的设备由参数hDC指定；
 *绘制的矩形区域由参数lpDCRect指定；输出DIB的区域由参数
 * lpDIBRect指定。
 *
 ************************************************************************/

BOOL WINAPI PaintDIB(HDC        hDC,
                     LPRECT     lpDCRect,
                     HDIB       hDIB,
                     LPRECT     lpDIBRect,
                     CPalette* pPal)
{

    LPSTR    lpDIBHdr;              // BITMAPINFOHEADER指针
    LPSTR    lpDIBBits;             // DIB像素指针
    BOOL     bSuccess=FALSE;        // 成功标志
    HPALETTE hPal=NULL;             // DIB调色板
    HPALETTE hOldPal=NULL;          // 以前的调色板

    // 判断DIB对象是否为空
    if (hDIB == NULL)
    {
        // 返回
        return FALSE;
    }

    // 锁定DIB
    lpDIBHdr   = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIBHdr);

    // 获取DIB调色板，并选中它
    if (pPal != NULL)
    {
        hPal = (HPALETTE) pPal->m_hObject;
```

```
        // 选中调色板
        hOldPal = ::SelectPalette(hDC, hPal, TRUE);
    }

    // 设置显示模式
    ::SetStretchBltMode(hDC, COLORONCOLOR);

    // 判断是调用StretchDIBits()还是SetDIBitsToDevice()来绘制DIB对象
    if ((RECTWIDTH(lpDCRect)  == RECTWIDTH(lpDIBRect)) &&
       (RECTHEIGHT(lpDCRect) == RECTHEIGHT(lpDIBRect)))
    {
        // 原始大小，不用拉伸。
        bSuccess = ::SetDIBitsToDevice(hDC,                    // hDC
                                  lpDCRect->left,              // DestX
                                  lpDCRect->top,               // DestY
                                  RECTWIDTH(lpDCRect),         // nDestWidth
                                  RECTHEIGHT(lpDCRect),        // nDestHeight
                                  lpDIBRect->left,             // SrcX
                                  (int)DIBHeight(lpDIBHdr) -
                                   lpDIBRect->top -
                                   RECTHEIGHT(lpDIBRect),      // SrcY
                                  0,                           // nStartScan
                                  (WORD)DIBHeight(lpDIBHdr),   // nNumScans
                                  lpDIBBits,                   // lpBits
                                  (LPBITMAPINFO)lpDIBHdr,      // lpBitsInfo
                                  DIB_RGB_COLORS);             // wUsage
    }
    else
    {
        // 非原始大小，拉伸。
        bSuccess = ::StretchDIBits(hDC,                        // hDC
                              lpDCRect->left,                  // DestX
                              lpDCRect->top,                   // DestY
                              RECTWIDTH(lpDCRect),             // nDestWidth
                              RECTHEIGHT(lpDCRect),            // nDestHeight
                              lpDIBRect->left,                 // SrcX
                              lpDIBRect->top,                  // SrcY
                              RECTWIDTH(lpDIBRect),            // wSrcWidth
                              RECTHEIGHT(lpDIBRect),           // wSrcHeight
                              lpDIBBits,                       // lpBits
                              (LPBITMAPINFO)lpDIBHdr,          // lpBitsInfo
                              DIB_RGB_COLORS,                  // wUsage
                              SRCCOPY);                        // dwROP
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) hDIB);

    // 恢复以前的调色板
    if (hOldPal != NULL)
    {
```

```
        ::SelectPalette(hDC, hOldPal, TRUE);
    }

    // 返回
    return bSuccess;
}

/*************************************************************************
 *
 * 函数名称：
 *   CreateDIBPalette()
 *
 * 参数:
 *   HDIB hDIB            - 指向DIB对象的指针
 *   CPalette* pPal       - 指向DIB对象调色板的指针
 *
 * 返回值:
 *   BOOL                 - 创建成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数按照DIB创建一个逻辑调色板，从DIB中读取颜色表并存到调色板中，
 * 最后按照该逻辑调色板创建一个新的调色板，并返回该调色板的句柄。这样
 * 可以用最好的颜色来显示DIB图像。
 *
 ************************************************************************/


BOOL WINAPI CreateDIBPalette(HDIB hDIB, CPalette* pPal)
{

    // 指向逻辑调色板的指针
    LPLOGPALETTE lpPal;

    // 逻辑调色板的句柄
    HANDLE hLogPal;

    // 调色板的句柄
    HPALETTE hPal = NULL;

    // 循环变量
    int i;

    // 颜色表中的颜色数目
    WORD wNumColors;

    // 指向DIB的指针
    LPSTR lpbi;

    // 指向BITMAPINFO结构的指针（Win3.0）
    LPBITMAPINFO lpbmi;
```

```
// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREINFO lpbmc;

// 表明是否是Win3.0 DIB的标记
BOOL bWinStyleDIB;

// 创建结果
BOOL bResult = FALSE;

// 判断DIB是否为空
if (hDIB == NULL)
{
    // 返回FALSE
    return FALSE;
}

// 锁定DIB
lpbi = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

// 获取指向BITMAPINFO结构的指针（Win3.0）
lpbmi = (LPBITMAPINFO)lpbi;

// 获取指向BITMAPCOREINFO结构的指针
lpbmc = (LPBITMAPCOREINFO)lpbi;

// 获取DIB中颜色表中的颜色数目
wNumColors = ::DIBNumColors(lpbi);

if (wNumColors != 0)
{
    // 分配为逻辑调色板内存
    hLogPal = ::GlobalAlloc(GHND, sizeof(LOGPALETTE)
                            + sizeof(PALETTEENTRY)
                            * wNumColors);

    // 如果内存不足，退出
    if (hLogPal == 0)
    {
        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDIB);

        // 返回FALSE
        return FALSE;
    }

    lpPal = (LPLOGPALETTE) ::GlobalLock((HGLOBAL) hLogPal);

    // 设置版本号
    lpPal->palVersion = PALVERSION;
```

```
        // 设置颜色数目
        lpPal->palNumEntries = (WORD)wNumColors;

        // 判断是否是WIN3.0的DIB
        bWinStyleDIB = IS_WIN30_DIB(lpbi);

        // 读取调色板
        for (i = 0; i < (int)wNumColors; i++)
        {
            if (bWinStyleDIB)
            {
                // 读取红色分量
                lpPal->palPalEntry[i].peRed = lpbmi->bmiColors[i].rgbRed;

                // 读取绿色分量
                lpPal->palPalEntry[i].peGreen = lpbmi->bmiColors[i].rgbGreen;

                // 读取蓝色分量
                lpPal->palPalEntry[i].peBlue = lpbmi->bmiColors[i].rgbBlue;

                // 保留位
                lpPal->palPalEntry[i].peFlags = 0;
            }
            else
            {
                // 读取红色分量
                lpPal->palPalEntry[i].peRed = lpbmc->bmciColors[i].rgbtRed;

                // 读取绿色分量
                lpPal->palPalEntry[i].peGreen = lpbmc->bmciColors[i].rgbtGreen;

                // 读取红色分量
                lpPal->palPalEntry[i].peBlue = lpbmc->bmciColors[i].rgbtBlue;

                // 保留位
                lpPal->palPalEntry[i].peFlags = 0;
            }
        }

        // 按照逻辑调色板创建调色板，并返回指针
        bResult = pPal->CreatePalette(lpPal);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hLogPal);

        // 释放逻辑调色板
        ::GlobalFree((HGLOBAL) hLogPal);
    }

    // 解除锁定
```

```
    ::GlobalUnlock((HGLOBAL) hDIB);

    // 返回结果
    return bResult;
}

/*************************************************************************
 *
 * 函数名称：
 *   FindDIBBits()
 *
 * 参数:
 *   LPSTR lpbi          - 指向DIB对象的指针
 *
 * 返回值:
 *   LPSTR               - 指向DIB图像像素起始位置
 *
 * 说明:
 *   该函数计算DIB中图像像素的起始位置，并返回指向它的指针。
 *
 ************************************************************************/


LPSTR WINAPI FindDIBBits(LPSTR lpbi)
{
    return (lpbi + *(LPDWORD)lpbi + ::PaletteSize(lpbi));
}


/*************************************************************************
 *
 * 函数名称：
 *   DIBWidth()
 *
 * 参数:
 *   LPSTR lpbi          - 指向DIB对象的指针
 *
 * 返回值:
 *   DWORD               - DIB中图像的宽度
 *
 * 说明:
 *   该函数返回DIB中图像的宽度。对于Windows 3.0 DIB，返回BITMAPINFOHEADER
 * 中的biWidth值；对于其他返回BITMAPCOREHEADER中的bcWidth值。
 *
 ************************************************************************/


DWORD WINAPI DIBWidth(LPSTR lpDIB)
{

    // 指向BITMAPINFO结构的指针（Windows 3.0）
```

```
    LPBITMAPINFOHEADER lpbmi;

    // 指向BITMAPCOREINFO结构的指针
    LPBITMAPCOREHEADER lpbmc;

    // 获取指针
    lpbmi = (LPBITMAPINFOHEADER)lpDIB;
    lpbmc = (LPBITMAPCOREHEADER)lpDIB;

    // 返回DIB中图像的宽度
    if (IS_WIN30_DIB(lpDIB))
    {
        // 对于Windows 3.0 DIB，返回lpbmi->biWidth
        return lpbmi->biWidth;
    }
    else
    {
        // 对于其他格式的DIB，返回lpbmc->bcWidth
        return (DWORD)lpbmc->bcWidth;
    }
}


/*************************************************************************
 *
 * 函数名称:
 *   DIBHeight()
 *
 * 参数:
 *   LPSTR lpbi          - 指向DIB对象的指针
 *
 * 返回值:
 *   DWORD               - DIB中图像的高度
 *
 * 说明:
 *   该函数返回DIB中图像的高度。对于Windows 3.0 DIB，返回BITMAPINFOHEADER
 * 中的biHeight值：对于其他返回BITMAPCOREHEADER中的bcHeight值。
 *
 ************************************************************************/


DWORD WINAPI DIBHeight(LPSTR lpDIB)
{

    // 指向BITMAPINFO结构的指针（Win3.0）
    LPBITMAPINFOHEADER lpbmi;

    // 指向BITMAPCOREINFO结构的指针
    LPBITMAPCOREHEADER lpbmc;
```

```
    // 获取指针
    lpbmi = (LPBITMAPINFOHEADER)lpDIB;
    lpbmc = (LPBITMAPCOREHEADER)lpDIB;

    // 返回DIB中图像的宽度
    if (IS_WIN30_DIB(lpDIB))
    {
        // 对于Windows 3.0 DIB，返回lpbmi->biHeight
        return lpbmi->biHeight;
    }
    else
    {
        // 对于其他格式的DIB，返回lpbmc->bcHeight
        return (DWORD)lpbmc->bcHeight;
    }
}


/*************************************************************************
 *
 * 函数名称：
 *   PaletteSize()
 *
 * 参数:
 *   LPSTR lpbi          - 指向DIB对象的指针
 *
 * 返回值:
 *   WORD                - DIB中调色板的大小
 *
 * 说明:
 *   该函数返回DIB中调色板的大小。对于Windows 3.0 DIB，返回颜色数目×
 * RGBQUAD的大小；对于其他返回颜色数目×RGBTRIPLE的大小。
 *
 *************************************************************************/


WORD WINAPI PaletteSize(LPSTR lpbi)
{

    // 计算DIB中调色板的大小
    if (IS_WIN30_DIB (lpbi))
    {
        //返回颜色数目×RGBQUAD的大小
        return (WORD)(::DIBNumColors(lpbi) * sizeof(RGBQUAD));
    }
    else
    {
        //返回颜色数目×RGBTRIPLE的大小
        return (WORD)(::DIBNumColors(lpbi) * sizeof(RGBTRIPLE));
    }
}
```

```
/*************************************************************************
 *
 * 函数名称：
 *   DIBNumColors()
 *
 * 参数:
 *   LPSTR lpbi          - 指向DIB对象的指针
 *
 * 返回值:
 *   WORD                - 返回调色板中颜色的种数
 *
 * 说明:
 *   该函数返回DIB中调色板的颜色的种数。对于单色位图，返回2；
 * 对于16色位图，返回16；对于256色位图，返回256；对于真彩色
 * 位图（24位），没有调色板，返回0。
 *
 ************************************************************************/

WORD WINAPI DIBNumColors(LPSTR lpbi)
{
    WORD wBitCount;

    // 对于Windows的DIB, 实际颜色的数目可以比像素的位数要少。
    // 对于这种情况，则返回一个近似的数值。

    // 判断是否是WIN3.0 DIB
    if (IS_WIN30_DIB(lpbi))
    {
        DWORD dwClrUsed;

        // 读取dwClrUsed值
        dwClrUsed = ((LPBITMAPINFOHEADER)lpbi)->biClrUsed;

        if (dwClrUsed != 0)
        {
            // 如果dwClrUsed（实际用到的颜色数）不为0，直接返回该值。
            return (WORD)dwClrUsed;
        }
    }

    // 读取像素的位数
    if (IS_WIN30_DIB(lpbi))
    {
        // 读取biBitCount值
        wBitCount = ((LPBITMAPINFOHEADER)lpbi)->biBitCount;
    }
    else
```

```
    {
        // 读取biBitCount值
        wBitCount = ((LPBITMAPCOREHEADER)lpbi)->bcBitCount;
    }

    // 按照像素的位数计算颜色数目
    switch (wBitCount)
    {
        case 1:
            return 2;

        case 4:
            return 16;

        case 8:
            return 256;

        default:
            return 0;
    }
}


/*************************************************************************
 *
 * 函数名称:
 *   CopyHandle()
 *
 * 参数:
 *   HGLOBAL h              - 要复制的内存区域
 *
 * 返回值:
 *   HGLOBAL                - 复制后的新内存区域
 *
 * 说明:
 *   该函数复制指定的内存区域。返回复制后的新内存区域，出错时返回0。
 *
 ************************************************************************/

HGLOBAL WINAPI CopyHandle (HGLOBAL h)
{
    if (h == NULL)
        return NULL;

    // 获取指定内存区域大小
    DWORD dwLen = ::GlobalSize((HGLOBAL) h);

    // 分配新内存空间
    HGLOBAL hCopy = ::GlobalAlloc(GHND, dwLen);

    // 判断分配是否成功
```

```
    if (hCopy != NULL)
    {
        // 锁定
        void* lpCopy = ::GlobalLock((HGLOBAL) hCopy);
        void* lp     = ::GlobalLock((HGLOBAL) h);

        // 复制
        memcpy(lpCopy, lp, dwLen);

        // 解除锁定
        ::GlobalUnlock(hCopy);
        ::GlobalUnlock(h);
    }

    return hCopy;
}



/*************************************************************************
 *
 * 函数名称:
 *   SaveDIB()
 *
 * 参数:
 *   HDIB hDib           - 要保存的DIB
 *   CFile& file         - 保存文件CFile
 *
 * 返回值:
 *   BOOL                - 成功返回TRUE，否则返回FALSE或者CfileException。
 *
 * 说明:
 *   该函数将指定的DIB对象保存到指定的CFile中。该CFile由调用程序打开和关闭。
 *
 *************************************************************************/


BOOL WINAPI SaveDIB(HDIB hDib, CFile& file)
{
    // Bitmap文件头
    BITMAPFILEHEADER bmfHdr;

    // 指向BITMAPINFOHEADER的指针
    LPBITMAPINFOHEADER lpBI;

    // DIB大小
    DWORD dwDIBSize;

    if (hDib == NULL)
    {
```

```
        // 如果DIB为空，返回FALSE
        return FALSE;
    }

    // 读取BITMAPINFO结构，并锁定
    lpBI = (LPBITMAPINFOHEADER) ::GlobalLock((HGLOBAL) hDib);

    if (lpBI == NULL)
    {
        // 为空，返回FALSE
        return FALSE;
    }

    // 判断是否是WIN3.0 DIB
    if (!IS_WIN30_DIB(lpBI))
    {
        // 不支持其他类型的DIB保存

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDib);

        // 返回FALSE
        return FALSE;
    }

    // 填充文件头

    // 文件类型"BM"
    bmfHdr.bfType = DIB_HEADER_MARKER;

    // 计算DIB大小时，最简单的方法是调用GlobalSize()函数。但是全局内存大小并
    // 不是DIB真正的大小，它总是多几个字节。这样就需要计算一下DIB的真实大小。

    // 文件头大小＋颜色表大小
    // （BITMAPINFOHEADER和BITMAPCOREHEADER结构的第一个DWORD都是该结构的大
小）
    dwDIBSize = *(LPDWORD)lpBI + ::PaletteSize((LPSTR)lpBI);

    // 计算图像大小
    if ((lpBI->biCompression == BI_RLE8) || (lpBI->biCompression == BI_RLE4))
    {
        // 对于RLE位图，没法计算大小，只能信任biSizeImage内的值
        dwDIBSize += lpBI->biSizeImage;
    }
    else
    {
        // 像素的大小
        DWORD dwBmBitsSize;

        // 大小为Width * Height
        dwBmBitsSize = WIDTHBYTES((lpBI->biWidth)*((DWORD)lpBI->biBitCount)) *
```

```
lpBI->biHeight;

            // 计算出DIB真正的大小
            dwDIBSize += dwBmBitsSize;

            // 更新biSizeImage（很多BMP文件头中biSizeImage的值是错误的）
            lpBI->biSizeImage = dwBmBitsSize;
        }


        // 计算文件大小：DIB大小+BITMAPFILEHEADER结构大小
        bmfHdr.bfSize = dwDIBSize + sizeof(BITMAPFILEHEADER);

        // 两个保留字
        bmfHdr.bfReserved1 = 0;
        bmfHdr.bfReserved2 = 0;

        // 计算偏移量bfOffBits，它的大小为Bitmap文件头大小+DIB头大小+颜色表大小
        bmfHdr.bfOffBits = (DWORD)sizeof(BITMAPFILEHEADER) + lpBI->biSize
                                                  + PaletteSize((LPSTR)lpBI);
        // 尝试写文件
        TRY
        {
            // 写文件头
            file.Write((LPSTR)&bmfHdr, sizeof(BITMAPFILEHEADER));

            // 写DIB头和像素
            file.WriteHuge(lpBI, dwDIBSize);
        }
        CATCH (CFileException, e)
        {
            // 解除锁定
            ::GlobalUnlock((HGLOBAL) hDib);

            // 抛出异常
            THROW_LAST();
        }
        END_CATCH

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDib);

        // 返回TRUE
        return TRUE;
    }


    /*************************************************************************
     *
     * 函数名称：
```

```
 *   ReadDIBFile()
 *
 * 参数:
 *   CFile& file            - 要读取得文件文件CFile
 *
 * 返回值:
 *   HDIB                   - 成功返回DIB的句柄，否则返回NULL。
 *
 * 说明:
 *   该函数将指定的文件中的DIB对象读到指定的内存区域中，除BITMAPFILEHEADER
 * 外的内容都将被读入内存。
 *
 ************************************************************************/

HDIB WINAPI ReadDIBFile(CFile& file)
{
    BITMAPFILEHEADER bmfHeader;
    DWORD dwBitsSize;
    HDIB hDIB;
    LPSTR pDIB;

    // 获取DIB（文件）长度（字节）
    dwBitsSize = file.GetLength();

    // 尝试读取DIB文件头
    if (file.Read((LPSTR)&bmfHeader, sizeof(bmfHeader)) != sizeof(bmfHeader))
    {
        // 大小不对，返回NULL
        return NULL;
    }

    // 判断是否是DIB对象，检查头两个字节是否是“BM”
    if (bmfHeader.bfType != DIB_HEADER_MARKER)
    {
        // 非DIB对象，返回NULL
        return NULL;
    }

    // 为DIB分配内存
    hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, dwBitsSize);
    if (hDIB == 0)
    {
        // 内存分配失败，返回NULL
        return NULL;
    }

    // 锁定
    pDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

    // 读像素
    if (file.ReadHuge(pDIB, dwBitsSize - sizeof(BITMAPFILEHEADER)) !=
```

```
            dwBitsSize - sizeof(BITMAPFILEHEADER) )
        {
            // 大小不对

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) hDIB);

            // 释放内存
            ::GlobalFree((HGLOBAL) hDIB);

            // 返回NULL
            return NULL;
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDIB);


        // 返回DIB句柄
        return hDIB;
    }
```

本书后面的示例程序中将直接用到这些 DIB 函数。

2.3.4 使用 DIB 读写 BMP 文件示例

下面我们就使用这些 DIB 函数来编写一个简单的读写 BMP 的多文档示例。该示例不但可以直接读写 BMP 文件，打印当前 DIB，还支持剪贴板操作：复制当前 DIB 到剪贴板，也可以将剪贴板中现有的 DIB 拷贝到当前的 DIB 中。

程序运行如图 2－8 所示。



图 2－8　读写 BMP 文件应用程序示例图

下面是源代码清单（工程名称为 ch1_1)。注意：本书代码中阴影部分为读者需要注意的地方。

1. ch1_1.cpp 源代码

```
// ch1_1.cpp : Defines the class behaviors for the application.
//

#include "stdafx.h"
#include "ch1_1.h"

#include "MainFrm.h"
#include "ChildFrm.h"

#include "ch1_1Doc.h"
#include "ch1_1View.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CCh1_1App

BEGIN_MESSAGE_MAP(CCh1_1App, CWinApp)
    //{{AFX_MSG_MAP(CCh1_1App)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG_MAP
    // Standard file based document commands
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    // Standard print setup command
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()


/////////////////////////////////////////////////////////////////////////////
// CCh1_1App construction

CCh1_1App::CCh1_1App()
{
    // TODO: add construction code here,
    // Place all significant initialization in InitInstance
}

/////////////////////////////////////////////////////////////////////////////
// The one and only CCh1_1App object
```

```
CCh1_1App theApp;

/////////////////////////////////////////////////////////////////////////////
// CCh1_1App initialization

BOOL CCh1_1App::InitInstance()
{
    AfxEnableControlContainer();

    // Standard initialization
    // If you are not using these features and wish to reduce the size
    //  of your final executable, you should remove from the following
    //  the specific initialization routines you do not need.

#ifdef _AFXDLL
    Enable3dControls();             // Call this when using MFC in a shared DLL
#else
    Enable3dControlsStatic();   // Call this when linking to MFC statically
#endif

    // Change the registry key under which our settings are stored.
    // TODO: You should modify this string to be something appropriate
    // such as the name of your company or organization.
    SetRegistryKey(_T("Local AppWizard-Generated Applications"));

    LoadStdProfileSettings();  // Load standard INI file options (including MRU)


    // Register the application's document templates.  Document templates
    //  serve as the connection between documents, frame windows and views.

    CMultiDocTemplate* pDocTemplate;

    pDocTemplate = new CMultiDocTemplate(
        IDR_CH1_1TYPE,
        RUNTIME_CLASS(CCh1_1Doc),
        RUNTIME_CLASS(CChildFrame), // custom MDI child frame
        RUNTIME_CLASS(CCh1_1View));
    AddDocTemplate(pDocTemplate);

    // create main MDI Frame window
    CMainFrame* pMainFrame = new CMainFrame;
    if (!pMainFrame->LoadFrame(IDR_MAINFRAME))
        return FALSE;
    m_pMainWnd = pMainFrame;

    // Parse command line for standard shell commands, DDE, file open
    CCommandLineInfo cmdInfo;
    ParseCommandLine(cmdInfo);
```

```
    // 启动时不自动打开一个空文档
    cmdInfo.m_nShellCommand = CCommandLineInfo::FileNothing;

    // Dispatch commands specified on the command line
    if (!ProcessShellCommand(cmdInfo))
        return FALSE;

    // The main window has been initialized, so show and update it.
    pMainFrame->ShowWindow(SW_SHOWMAXIMIZED);
    pMainFrame->UpdateWindow();

    return TRUE;

}

/////////////////////////////////////////////////////////////////////////////
// CAboutDlg dialog used for App About

class CAboutDlg : public CDialog
{

public:
    CAboutDlg();

// Dialog Data
    //{{AFX_DATA(CAboutDlg)
    enum { IDD = IDD_ABOUTBOX };
    //}}AFX_DATA

    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CAboutDlg)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);      // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:
    //{{AFX_MSG(CAboutDlg)
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


CAboutDlg::CAboutDlg() : CDialog(CAboutDlg::IDD)
{
    //{{AFX_DATA_INIT(CAboutDlg)
    //}}AFX_DATA_INIT
}

void CAboutDlg::DoDataExchange(CDataExchange* pDX)
{
```

```
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CAboutDlg)
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CAboutDlg, CDialog)
    //{{AFX_MSG_MAP(CAboutDlg)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

// App command to run the dialog
void CCh1_1App::OnAppAbout()
{
    CAboutDlg aboutDlg;
    aboutDlg.DoModal();
}
```

2. ch1_1Doc.cpp 源代码

```
// ch1_1Doc.cpp : implementation of the CCh1_1Doc class
//

#include "stdafx.h"
#include "ch1_1.h"

#include "ch1_1Doc.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CCh1_1Doc

IMPLEMENT_DYNCREATE(CCh1_1Doc, CDocument)

BEGIN_MESSAGE_MAP(CCh1_1Doc, CDocument)
    //{{AFX_MSG_MAP(CCh1_1Doc)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    ON_COMMAND(ID_FILE_REOPEN, OnFileReopen)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCh1_1Doc construction/destruction

CCh1_1Doc::CCh1_1Doc()
{
```

```
    // 默认背景色，灰色
    m_refColorBKG = 0x00808080;

    // 初始化变量
    m_hDIB = NULL;
    m_palDIB = NULL;
    m_sizeDoc = CSize(1,1);
}

CCh1_1Doc::~CCh1_1Doc()
{
    // 判断DIB对象是否存在
    if (m_hDIB != NULL)
    {
        // 清除DIB对象
        ::GlobalFree((HGLOBAL) m_hDIB);
    }
    // 判断调色板是否存在
    if (m_palDIB != NULL)
    {
        // 清除调色板
        delete m_palDIB;
    }
}

BOOL CCh1_1Doc::OnNewDocument()
{
    if (!CDocument::OnNewDocument())
        return FALSE;

    // TODO: add reinitialization code here
    // (SDI documents will reuse this document)

    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CCh1_1Doc diagnostics

#ifdef _DEBUG
void CCh1_1Doc::AssertValid() const
{
    CDocument::AssertValid();
}

void CCh1_1Doc::Dump(CDumpContext& dc) const
{
    CDocument::Dump(dc);
}
#endif //_DEBUG
```

```
////////////////////////////////////////////////////////////////////////////
// CCh1_1Doc commands

BOOL CCh1_1Doc::CanCloseFrame(CFrameWnd* pFrame)
{
    // TODO: Add your specialized code here and/or call the base class

    return CDocument::CanCloseFrame(pFrame);
}

void CCh1_1Doc::DeleteContents()
{
    // TODO: Add your specialized code here and/or call the base class

    CDocument::DeleteContents();
}

BOOL CCh1_1Doc::OnOpenDocument(LPCTSTR lpszPathName)
{
    CFile file;
    CFileException fe;

    // 打开文件
    if (!file.Open(lpszPathName, CFile::modeRead | CFile::shareDenyWrite, &fe))
    {
        // 失败
        ReportSaveLoadException(lpszPathName, &fe,
            FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

        // 返回FALSE
        return FALSE;
    }

    DeleteContents();

    // 更改光标形状
    BeginWaitCursor();

    // 尝试调用ReadDIBFile()读取图像
    TRY
    {
        m_hDIB = ::ReadDIBFile(file);
    }
    CATCH (CFileException, eLoad)
    {
        // 读取失败
        file.Abort();

        // 恢复光标形状
        EndWaitCursor();
```

```
        // 报告失败
        ReportSaveLoadException(lpszPathName, eLoad,
            FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

        // 设置DIB为空
        m_hDIB = NULL;

        // 返回FALSE
        return FALSE;
    }
    END_CATCH

    // 初始化DIB
    InitDIBData();

    // 恢复光标形状
    EndWaitCursor();

    // 判断读取文件是否成功
    if (m_hDIB == NULL)
    {
        // 失败，可能非BMP格式
        CString strMsg;
        strMsg = "读取图像时出错！可能是不支持该类型的图像文件！";

        // 提示出错
        MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION | MB_OK);

        // 返回FALSE
        return FALSE;
    }

    // 设置文件名称
    SetPathName(lpszPathName);

    // 初始化脏标记为FALSE
    SetModifiedFlag(FALSE);

    // 返回TRUE
    return TRUE;
}

void CCh1_1Doc::OnFileReopen()
{
    // 重新打开图像，放弃所有修改

    // 判断当前图像是否已经被改动
    if (IsModified())
    {
        // 提示用户该操作将丢失所有当前的修改
```

```
            if (MessageBox(NULL, "重新打开图像将丢失所有改动！是否继续？", "系统提示",
MB_ICONQUESTION | MB_YESNO) == IDNO)
            {
                // 用户取消操作，直接返回
                return;
            }
        }
        CFile file;
        CFileException fe;

        CString strPathName;

        // 获取当前文件路径
        strPathName = GetPathName();

        // 重新打开文件
        if (!file.Open(strPathName, CFile::modeRead | CFile::shareDenyWrite, &fe))
        {
            // 失败
            ReportSaveLoadException(strPathName, &fe,
                FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

            // 返回
            return;
        }

        // 更改光标形状
        BeginWaitCursor();

        // 尝试调用ReadDIBFile()读取图像
        TRY
        {
            m_hDIB = ::ReadDIBFile(file);
        }
        CATCH (CFileException, eLoad)
        {
            // 读取失败
            file.Abort();

            // 恢复光标形状
            EndWaitCursor();

            // 报告失败
            ReportSaveLoadException(strPathName, eLoad,
                FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

            // 设置DIB为空
            m_hDIB = NULL;

            // 返回
```

```
        return;
    }
    END_CATCH

    // 初始化DIB
    InitDIBData();

    // 判断读取文件是否成功
    if (m_hDIB == NULL)
    {
        // 失败，可能非BMP格式
        CString strMsg;
        strMsg = "读取图像时出错！可能是不支持该类型的图像文件！";

        // 提示出错
        MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION | MB_OK);

        // 恢复光标形状
        EndWaitCursor();

        // 返回
        return;
    }

    // 初始化脏标记为FALSE
    SetModifiedFlag(FALSE);

    // 刷新
    UpdateAllViews(NULL);

    // 恢复光标形状
    EndWaitCursor();

    // 返回
    return;

}

BOOL CCh1_1Doc::OnSaveDocument(LPCTSTR lpszPathName)
{
    CFile file;
    CFileException fe;

    // 打开文件
    if (!file.Open(lpszPathName, CFile::modeCreate |
      CFile::modeReadWrite | CFile::shareExclusive, &fe))
    {
        // 失败
        ReportSaveLoadException(lpszPathName, &fe,
            TRUE, AFX_IDP_INVALID_FILENAME);
```

```
        // 返回FALSE
        return FALSE;
    }

    // 尝试调用SaveDIB保存图像
    BOOL bSuccess = FALSE;
    TRY
    {

        // 更改光标形状
        BeginWaitCursor();

        // 尝试保存图像
        bSuccess = ::SaveDIB(m_hDIB, file);

        // 关闭文件
        file.Close();
    }
    CATCH (CException, eSave)
    {
        // 失败
        file.Abort();

        // 恢复光标形状
        EndWaitCursor();
        ReportSaveLoadException(lpszPathName, eSave,
            TRUE, AFX_IDP_FAILED_TO_SAVE_DOC);

        // 返回FALSE
        return FALSE;
    }
    END_CATCH

    // 恢复光标形状
    EndWaitCursor();

    // 重置脏标记为FALSE
    SetModifiedFlag(FALSE);

    if (!bSuccess)
    {
        // 保存失败，可能是其他格式的DIB，可以读取但是不能保存
        // 或者是SaveDIB函数有误

        CString strMsg;
        strMsg = "无法保存BMP图像！";

        // 提示出错
        MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION | MB_OK);
    }
```

```
    return bSuccess;
}

void CCh1_1Doc::ReplaceHDIB(HDIB hDIB)
{
    // 替换DIB，在功能粘贴中用到该函数

    // 判断DIB是否为空
    if (m_hDIB != NULL)
    {
        // 非空，则清除
        ::GlobalFree((HGLOBAL) m_hDIB);
    }

    // 替换成新的DIB对象
    m_hDIB = hDIB;

}

void CCh1_1Doc::InitDIBData()
{
    // 初始化DIB对象

    // 判断调色板是否为空
    if (m_palDIB != NULL)
    {
        // 删除调色板对象
        delete m_palDIB;

        // 重置调色板为空
        m_palDIB = NULL;
    }

    // 如果DIB对象为空，直接返回
    if (m_hDIB == NULL)
    {
        // 返回
        return;
    }

    LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) m_hDIB);

    // 判断图像是否过大
    if (::DIBWidth(lpDIB) > INT_MAX ||::DIBHeight(lpDIB) > INT_MAX)
    {
        ::GlobalUnlock((HGLOBAL) m_hDIB);

        // 释放DIB对象
        ::GlobalFree((HGLOBAL) m_hDIB);
```

```
        // 设置DIB为空
        m_hDIB = NULL;

        CString strMsg;
        strMsg = "BMP图像太大！ ";

        // 提示用户
        MessageBox(NULL, strMsg, NULL, MB_ICONINFORMATION | MB_OK);

        // 返回
        return;
    }

    // 设置文档大小
    m_sizeDoc = CSize((int) ::DIBWidth(lpDIB), (int) ::DIBHeight(lpDIB));

    ::GlobalUnlock((HGLOBAL) m_hDIB);

    // 创建新调色板
    m_palDIB = new CPalette;

    // 判断是否创建成功
    if (m_palDIB == NULL)
    {
        // 失败，可能是内存不足
        ::GlobalFree((HGLOBAL) m_hDIB);

        // 设置DIB对象为空
        m_hDIB = NULL;

        // 返回
        return;
    }

    // 调用CreateDIBPalette来创建调色板
    if (::CreateDIBPalette(m_hDIB, m_palDIB) == NULL)
    {
        // 返回空，可能该DIB对象没有调色板

        // 删除
        delete m_palDIB;

        // 设置为空
        m_palDIB = NULL;

        // 返回
        return;
    }
}
```

3. ch1_1View.cpp 源代码

```
// ch1_1View.cpp : implementation of the CCh1_1View class
//

#include "stdafx.h"
#include "ch1_1.h"

#include "ch1_1Doc.h"
#include "ch1_1View.h"
#include "mainfrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View

IMPLEMENT_DYNCREATE(CCh1_1View, CScrollView)

BEGIN_MESSAGE_MAP(CCh1_1View, CScrollView)
    //{{AFX_MSG_MAP(CCh1_1View)
    ON_WM_ERASEBKGND()
    ON_COMMAND(ID_EDIT_COPY, OnEditCopy)
    ON_COMMAND(ID_EDIT_PASTE, OnEditPaste)
    ON_UPDATE_COMMAND_UI(ID_EDIT_COPY, OnUpdateEditCopy)
    ON_UPDATE_COMMAND_UI(ID_EDIT_PASTE, OnUpdateEditPaste)
    //}}AFX_MSG_MAP
    // Standard printing commands
    ON_COMMAND(ID_FILE_PRINT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_DIRECT, CScrollView::OnFilePrint)
    ON_COMMAND(ID_FILE_PRINT_PREVIEW, CScrollView::OnFilePrintPreview)
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View construction/destruction

CCh1_1View::CCh1_1View()
{
    // TODO: add construction code here

}

CCh1_1View::~CCh1_1View()
{
}

BOOL CCh1_1View::PreCreateWindow(CREATESTRUCT& cs)
{
    // TODO: Modify the Window class or styles here by modifying
```

```
    //  the CREATESTRUCT cs

    return CView::PreCreateWindow(cs);
}

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View drawing

void CCh1_1View::OnDraw(CDC* pDC)
{

    // 显示等待光标
    BeginWaitCursor();

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();
    ASSERT_VALID(pDoc);

    // 获取DIB
    HDIB hDIB = pDoc->GetHDIB();

    // 判断DIB是否为空
    if (hDIB != NULL)
    {
        LPSTR lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

        // 获取DIB宽度
        int cxDIB = (int) ::DIBWidth(lpDIB);

        // 获取DIB高度
        int cyDIB = (int) ::DIBHeight(lpDIB);

        ::GlobalUnlock((HGLOBAL) hDIB);

        CRect rcDIB;
        rcDIB.top = rcDIB.left = 0;
        rcDIB.right = cxDIB;
        rcDIB.bottom = cyDIB;

        CRect rcDest;

        // 判断是否是打印
        if (pDC->IsPrinting())
        {
            // 是打印，计算输出图像的位置和大小以符合页面

            // 获取打印页面的水平宽度(像素)
            int cxPage = pDC->GetDeviceCaps(HORZRES);

            // 获取打印页面的垂直高度(像素)
```

```
            int cyPage = pDC->GetDeviceCaps(VERTRES);

            // 获取打印机每英寸像素数
            int cxInch = pDC->GetDeviceCaps(LOGPIXELSX);
            int cyInch = pDC->GetDeviceCaps(LOGPIXELSY);

            // 计算打印图像大小（缩放，根据页面宽度调整图像大小）
            rcDest.top = rcDest.left = 0;
            rcDest.bottom = (int)(((double)cyDIB * cxPage * cyInch)
                    / ((double)cxDIB * cxInch));
            rcDest.right = cxPage;

            // 计算打印图像位置（垂直居中）
            int temp = cyPage - (rcDest.bottom - rcDest.top);
            rcDest.bottom += temp/2;
            rcDest.top += temp/2;
        }
        else
        // 非打印
        {
            // 不必缩放图像
            rcDest = rcDIB;
        }

        // 输出DIB
        ::PaintDIB(pDC->m_hDC, &rcDest, pDoc->GetHDIB(),
            &rcDIB, pDoc->GetDocPalette());
    }

    // 恢复正常光标
    EndWaitCursor();

}

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View printing

BOOL CCh1_1View::OnPreparePrinting(CPrintInfo* pInfo)
{
    // 设置总页数为1。
    pInfo->SetMaxPage(1);

    return DoPreparePrinting(pInfo);
}

void CCh1_1View::OnBeginPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
{
    // TODO: add extra initialization before printing
}

void CCh1_1View::OnEndPrinting(CDC* /*pDC*/, CPrintInfo* /*pInfo*/)
```

```
{
    // TODO: add cleanup after printing
}

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View diagnostics

#ifdef _DEBUG
void CCh1_1View::AssertValid() const
{
    CView::AssertValid();
}

void CCh1_1View::Dump(CDumpContext& dc) const
{
    CView::Dump(dc);
}

CCh1_1Doc* CCh1_1View::GetDocument() // non-debug version is inline
{
    ASSERT(m_pDocument->IsKindOf(RUNTIME_CLASS(CCh1_1Doc)));
    return (CCh1_1Doc*)m_pDocument;
}
#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CCh1_1View message handlers


BOOL CCh1_1View::OnEraseBkgnd(CDC* pDC)
{
    // 主要是为了设置子窗体默认的背景色
    // 背景色由文档成员变量m_refColorBKG指定

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 创建一个Brush
    CBrush brush(pDoc->m_refColorBKG);

    // 保存以前的Brush
    CBrush* pOldBrush = pDC->SelectObject(&brush);

    // 获取重绘区域
    CRect rectClip;
    pDC->GetClipBox(&rectClip);

    // 重绘
    pDC->PatBlt(rectClip.left, rectClip.top, rectClip.Width(), rectClip.Height(), PATCOPY);
```

```
    // 恢复以前的Brush
    pDC->SelectObject(pOldBrush);

    // 返回
    return TRUE;

}

LRESULT CCh1_1View::OnDoRealize(WPARAM wParam, LPARAM)
{
    ASSERT(wParam != NULL);

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 判断DIB是否为空
    if (pDoc->GetHDIB() == NULL)
    {
        // 直接返回
        return 0L;
    }

    // 获取Palette
    CPalette* pPal = pDoc->GetDocPalette();
    if (pPal != NULL)
    {
        // 获取MainFrame
        CMainFrame* pAppFrame = (CMainFrame*) AfxGetApp()->m_pMainWnd;
        ASSERT_KINDOF(CMainFrame, pAppFrame);

        CClientDC appDC(pAppFrame);

        // All views but one should be a background palette.
        // wParam contains a handle to the active view, so the SelectPalette
        // bForceBackground flag is FALSE only if wParam == m_hWnd (this view)
        CPalette* oldPalette = appDC.SelectPalette(pPal, ((HWND)wParam) != m_hWnd);

        if (oldPalette != NULL)
        {
            UINT nColorsChanged = appDC.RealizePalette();
            if (nColorsChanged > 0)
                pDoc->UpdateAllViews(NULL);
            appDC.SelectPalette(oldPalette, TRUE);
        }
        else
        {
            TRACE0("\tCCh1_1View::OnPaletteChanged中调用SelectPalette()失败！ \n");
        }
    }

    return 0L;
```

```
}

void CCh1_1View::OnInitialUpdate()
{
    CView::OnInitialUpdate();

    // TODO: Add your specialized code here and/or call the base class

}

void CCh1_1View::CalcWindowRect(LPRECT lpClientRect, UINT nAdjustType)
{
    CScrollView::OnInitialUpdate();
    ASSERT(GetDocument() != NULL);

    SetScrollSizes(MM_TEXT, GetDocument()->GetDocSize());
}

void CCh1_1View::OnActivateView(BOOL bActivate, CView* pActivateView,
                    CView* pDeactiveView)
{
    CScrollView::OnActivateView(bActivate, pActivateView, pDeactiveView);

    if (bActivate)
    {
        ASSERT(pActivateView == this);
        OnDoRealize((WPARAM)m_hWnd, 0);    // same as SendMessage(WM_DOREALIZE);
    }
}

void CCh1_1View::OnEditCopy()
{
    // 复制当前图像

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 打开剪贴板
    if (OpenClipboard())
    {
        // 更改光标形状
        BeginWaitCursor();

        // 清空剪贴板
        EmptyClipboard();

        // 复制当前图像到剪贴板
        SetClipboardData (CF_DIB, CopyHandle((HANDLE) pDoc->GetHDIB()) );

        // 关闭剪贴板
```

```
        CloseClipboard();

        // 恢复光标
        EndWaitCursor();
    }
}

void CCh1_1View::OnEditPaste()
{
    // 粘贴图像

    // 创建新DIB
    HDIB hNewDIB = NULL;

    // 打开剪贴板
    if (OpenClipboard())
    {
        // 更改光标形状
        BeginWaitCursor();

        // 读取剪贴板中的图像
        hNewDIB = (HDIB) CopyHandle(::GetClipboardData(CF_DIB));

        // 关闭剪贴板
        CloseClipboard();

        // 判断是否读取成功
        if (hNewDIB != NULL)
        {
            // 获取文档
            CCh1_1Doc* pDoc = GetDocument();

            // 替换DIB，同时释放旧DIB对象
            pDoc->ReplaceHDIB(hNewDIB);

            // 更新DIB大小和调色板
            pDoc->InitDIBData();

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 重新设置滚动视图大小
            SetScrollSizes(MM_TEXT, pDoc->GetDocSize());

            // 实现新的调色板
            OnDoRealize((WPARAM)m_hWnd,0);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        // 恢复光标
```

```
            EndWaitCursor();
        }
    }

    void CCh1_1View::OnUpdateEditCopy(CCmdUI* pCmdUI)
    {
        // 如果当前DIB对象不空，复制菜单项有效
        pCmdUI->Enable(GetDocument()->GetHDIB() != NULL);
    }

    void CCh1_1View::OnUpdateEditPaste(CCmdUI* pCmdUI)
    {
        // 如果当前剪贴板中有DIB对象，粘贴菜单项有效
        pCmdUI->Enable(::IsClipboardFormatAvailable(CF_DIB));
    }
```

4. ChildFrm.cpp 源代码

```
// ChildFrm.cpp : implementation of the CChildFrame class
//

#include "stdafx.h"
#include "ch1_1.h"

#include "ChildFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CChildFrame

IMPLEMENT_DYNCREATE(CChildFrame, CMDIChildWnd)

BEGIN_MESSAGE_MAP(CChildFrame, CMDIChildWnd)
    //{{AFX_MSG_MAP(CChildFrame)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CChildFrame construction/destruction

CChildFrame::CChildFrame()
{
    // TODO: add member initialization code here
```

```
}

CChildFrame::~CChildFrame()
{
}

BOOL CChildFrame::PreCreateWindow(CREATESTRUCT& cs)
{

    // TODO: Modify the Window class or styles here by modifying
    //   the CREATESTRUCT cs

    if( !CMDIChildWnd::PreCreateWindow(cs) )
        return FALSE;

    cs.style = WS_CHILD | WS_VISIBLE | WS_OVERLAPPED | WS_CAPTION | WS_SYSMENU
        | FWS_ADDTOTITLE | WS_THICKFRAME | WS_MINIMIZEBOX | WS_MAXIMIZEBOX;

    return TRUE;
}

void CChildFrame::ActivateFrame(int nCmdShow)
{
    // TODO: Modify this function to change how the frame is activated.

    nCmdShow = SW_SHOWMAXIMIZED;
    CMDIChildWnd::ActivateFrame(nCmdShow);
}


/////////////////////////////////////////////////////////////////////////////
// CChildFrame diagnostics

#ifdef _DEBUG
void CChildFrame::AssertValid() const
{
    CMDIChildWnd::AssertValid();
}

void CChildFrame::Dump(CDumpContext& dc) const
{
    CMDIChildWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CChildFrame message handlers
```

5. MainFrm.cpp 源代码

```
// MainFrm.cpp : implementation of the CMainFrame class
//

#include "stdafx.h"
#include "ch1_1.h"

#include "MainFrm.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif


/////////////////////////////////////////////////////////////////////////////
// CMainFrame

IMPLEMENT_DYNAMIC(CMainFrame, CMDIFrameWnd)

BEGIN_MESSAGE_MAP(CMainFrame, CMDIFrameWnd)
    //{{AFX_MSG_MAP(CMainFrame)
        // NOTE - the ClassWizard will add and remove mapping macros here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    ON_WM_CREATE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

static UINT indicators[] =
{

    ID_SEPARATOR,           // status line indicator
    ID_INDICATOR_CAPS,
    ID_INDICATOR_NUM,
    ID_INDICATOR_SCRL,
};

/////////////////////////////////////////////////////////////////////////////
// CMainFrame construction/destruction

CMainFrame::CMainFrame()
{
    // TODO: add member initialization code here

}

CMainFrame::~CMainFrame()
{
}
```

```
int CMainFrame::OnCreate(LPCREATESTRUCT lpCreateStruct)
{

    if (CMDIFrameWnd::OnCreate(lpCreateStruct) == -1)
        return -1;

    if (!m_wndToolBar.CreateEx(this, TBSTYLE_FLAT, WS_CHILD | WS_VISIBLE | CBRS_TOP
        | CBRS_GRIPPER | CBRS_TOOLTIPS | CBRS_FLYBY | CBRS_SIZE_DYNAMIC) ||
        !m_wndToolBar.LoadToolBar(IDR_MAINFRAME))
    {
        TRACE0("Failed to create toolbar\n");
        return -1;        // fail to create
    }

    if (!m_wndStatusBar.Create(this) ||
        !m_wndStatusBar.SetIndicators(indicators,
          sizeof(indicators)/sizeof(UINT)))
    {
        TRACE0("Failed to create status bar\n");
        return -1;        // fail to create
    }

    // TODO: Delete these three lines if you don't want the toolbar to
    //   be dockable
    m_wndToolBar.EnableDocking(CBRS_ALIGN_ANY);
    EnableDocking(CBRS_ALIGN_ANY);
    DockControlBar(&m_wndToolBar);

    return 0;
}

BOOL CMainFrame::PreCreateWindow(CREATESTRUCT& cs)
{
    if( !CMDIFrameWnd::PreCreateWindow(cs) )
        return FALSE;
    // TODO: Modify the Window class or styles here by modifying
    //   the CREATESTRUCT cs

    cs.style = WS_OVERLAPPED | WS_CAPTION | FWS_ADDTOTITLE
        | WS_THICKFRAME | WS_SYSMENU | WS_MINIMIZEBOX | WS_MAXIMIZEBOX |
WS_MAXIMIZE;

    return TRUE;
}

/////////////////////////////////////////////////////////////////////////////
// CMainFrame diagnostics

#ifdef _DEBUG
void CMainFrame::AssertValid() const
{
```

```
    CMDIFrameWnd::AssertValid();
}

void CMainFrame::Dump(CDumpContext& dc) const
{
    CMDIFrameWnd::Dump(dc);
}

#endif //_DEBUG

/////////////////////////////////////////////////////////////////////////////
// CMainFrame message handlers
```

6. ch1_1.h 源代码

```
// ch1_1.h : main header file for the CH1_1 application
//

#if !defined(AFX_CH1_1_H__DD0FB619_2B6D_4BE0_9979_B6EF4A78EC25__INCLUDED_)
#define AFX_CH1_1_H__DD0FB619_2B6D_4BE0_9979_B6EF4A78EC25__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#ifndef __AFXWIN_H__
    #error include 'stdafx.h' before including this file for PCH
#endif

#include "resource.h"           // main symbols

/////////////////////////////////////////////////////////////////////////////
// CCh1_1App:
// See ch1_1.cpp for the implementation of this class
//

class CCh1_1App : public CWinApp
{

public:
    CCh1_1App();

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CCh1_1App)
    public:
    virtual BOOL InitInstance();
    //}}AFX_VIRTUAL

// Implementation
    //{{AFX_MSG(CCh1_1App)
```

```
    afx_msg void OnAppAbout();
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code !
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};


/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_CH1_1_H__DD0FB619_2B6D_4BE0_9979_B6EF4A78EC25__INCLUDED_)
```

7. ch1_1Doc.h 源代码

```
// ch1_1Doc.h : interface of the CCh1_1Doc class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_CH1_1DOC_H__766DD4EA_BF18_426A_BA52_B747D78F541C__INCLUDED_)
#define AFX_CH1_1DOC_H__766DD4EA_BF18_426A_BA52_B747D78F541C__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#include "dibapi.h"

class CCh1_1Doc : public CDocument
{

protected: // create from serialization only
    CCh1_1Doc();
    DECLARE_DYNCREATE(CCh1_1Doc)

// Attributes
public:
    HDIB GetHDIB() const
        { return m_hDIB; }
    CPalette* GetDocPalette() const
        { return m_palDIB; }
    CSize GetDocSize() const
        { return m_sizeDoc; }

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CCh1_1Doc)
```

```
	public:
	virtual BOOL OnNewDocument();
	virtual BOOL OnOpenDocument(LPCTSTR lpszPathName);
	virtual BOOL CanCloseFrame(CFrameWnd* pFrame);
	virtual BOOL OnSaveDocument(LPCTSTR lpszPathName);
	virtual void DeleteContents();
	//}}AFX_VIRTUAL

// Implementation
public:

	void ReplaceHDIB(HDIB hDIB);
	void InitDIBData();
	virtual ~CCh1_1Doc();

	// 背景色
	COLORREF m_refColorBKG;

#ifdef _DEBUG
	virtual void AssertValid() const;
	virtual void Dump(CDumpContext& dc) const;
#endif

protected:

// Generated message map functions
protected:
	//{{AFX_MSG(CCh1_1Doc)
		// NOTE - the ClassWizard will add and remove member functions here.
		//    DO NOT EDIT what you see in these blocks of generated code !
	afx_msg void OnFileReopen();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()

protected:

	HDIB m_hDIB;
	CPalette* m_palDIB;
	CSize m_sizeDoc;
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_CH1_1DOC_H__766DD4EA_BF18_426A_BA52_B747D78F541C__INCLUDED_)
```

8. ch1_1View.h 源代码

```
// ch1_1View.h : interface of the CCh1_1View class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_CH1_1VIEW_H__60AAD957_ED0B_48FF_834E_78C547411B15__INCLUDED_)
#define AFX_CH1_1VIEW_H__60AAD957_ED0B_48FF_834E_78C547411B15__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

class CCh1_1View : public CScrollView
{

protected: // create from serialization only
    CCh1_1View();
    DECLARE_DYNCREATE(CCh1_1View)

// Attributes
public:
    CCh1_1Doc* GetDocument();

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CCh1_1View)

    public:
    virtual void OnDraw(CDC* pDC);  // overridden to draw this view
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    virtual void OnInitialUpdate();
    virtual void OnActivateView(BOOL bActivate, CView* pActivateView,
                    CView* pDeactiveView);
    protected:
    virtual BOOL OnPreparePrinting(CPrintInfo* pInfo);
    virtual void OnBeginPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void OnEndPrinting(CDC* pDC, CPrintInfo* pInfo);
    virtual void CalcWindowRect(LPRECT lpClientRect, UINT nAdjustType = adjustBorder);

    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CCh1_1View();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif
```

```
protected:

// Generated message map functions
protected:
    //{{AFX_MSG(CCh1_1View)

    afx_msg BOOL OnEraseBkgnd(CDC* pDC);
    afx_msg void OnEditCopy();
    afx_msg void OnEditPaste();
    afx_msg void OnUpdateEditCopy(CCmdUI* pCmdUI);
    afx_msg void OnUpdateEditPaste(CCmdUI* pCmdUI);
    afx_msg LRESULT OnDoRealize(WPARAM wParam, LPARAM lParam);   // user message

    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

#ifndef _DEBUG   // debug version in ch1_1View.cpp
inline CCh1_1Doc* CCh1_1View::GetDocument()
    { return (CCh1_1Doc*)m_pDocument; }
#endif

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_CH1_1VIEW_H__60AAD957_ED0B_48FF_834E_78C547411B15__INCLUDED_)
```

9. ChildFrm.h 源代码

```
// ChildFrm.h : interface of the CChildFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_CHILDFRM_H__4B5DD00A_2541_4C9C_9E01_E9BDB74E9CB1__INCLUDED_
)
#define AFX_CHILDFRM_H__4B5DD00A_2541_4C9C_9E01_E9BDB74E9CB1__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000


class CChildFrame : public CMDIChildWnd
{
    DECLARE_DYNCREATE(CChildFrame)
public:
    CChildFrame();
```

```
// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CChildFrame)
    public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    virtual void ActivateFrame(int nCmdShow);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CChildFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

// Generated message map functions
protected:
    //{{AFX_MSG(CChildFrame)
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_CHILDFRM_H__4B5DD00A_2541_4C9C_9E01_E9BDB74E9CB1__INCLUDED_)
```

10. MainFrm.h 源代码

```
// MainFrm.h : interface of the CMainFrame class
//
/////////////////////////////////////////////////////////////////////////////

#if !defined(AFX_MAINFRM_H__E524EA1E_9F84_4E0E_8A57_C78D51325408__INCLUDED_)
#define AFX_MAINFRM_H__E524EA1E_9F84_4E0E_8A57_C78D51325408__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
```

```
class CMainFrame : public CMDIFrameWnd
{
    DECLARE_DYNAMIC(CMainFrame)
public:
    CMainFrame();

// Attributes
public:

// Operations
public:

// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CMainFrame)
    public:
    virtual BOOL PreCreateWindow(CREATESTRUCT& cs);
    //}}AFX_VIRTUAL

// Implementation
public:
    virtual ~CMainFrame();
#ifdef _DEBUG
    virtual void AssertValid() const;
    virtual void Dump(CDumpContext& dc) const;
#endif

protected:  // control bar embedded members
    CStatusBar  m_wndStatusBar;
    CToolBar    m_wndToolBar;

// Generated message map functions
protected:
    //{{AFX_MSG(CMainFrame)
    afx_msg int OnCreate(LPCREATESTRUCT lpCreateStruct);
        // NOTE - the ClassWizard will add and remove member functions here.
        //    DO NOT EDIT what you see in these blocks of generated code!
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

/////////////////////////////////////////////////////////////////////////////

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_MAINFRM_H__E524EA1E_9F84_4E0E_8A57_C78D51325408__INCLUDED_)
```

11. stdafx.cpp 源代码

```
// stdafx.cpp : source file that includes just the standard includes
//  ch1_1.pch will be the pre-compiled header
//  stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

12. stdafx.h 源代码

```
// stdafx.h : include file for standard system include files,
//  or project specific include files that are used frequently, but
//       are changed infrequently
//

#if !defined(AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_)
#define AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN        // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>         // MFC core and standard components
#include <afxext.h>         // MFC extensions
#include <afxdisp.h>        // MFC Automation classes
#include <afxdtctl.h>       // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>         // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_)
```

13. Resource.h 源代码

```
//{{NO_DEPENDENCIES}}
// Microsoft Visual C++ generated include file.
// Used by CH1_1.RC
//

#define IDD_ABOUTBOX            100
#define IDR_MAINFRAME           128
#define IDR_CH1_1TYPE           129

// Next default values for new objects
//
```

```
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_3D_CONTROLS              1
#define _APS_NEXT_RESOURCE_VALUE  130
#define _APS_NEXT_CONTROL_VALUE       1000
#define _APS_NEXT_SYMED_VALUE         101
#define _APS_NEXT_COMMAND_VALUE       32771
#endif
#endif
```

14. stdafx.cpp 源代码

```
// stdafx.cpp : source file that includes just the standard includes
//   ch1_1.pch will be the pre-compiled header
//   stdafx.obj will contain the pre-compiled type information

#include "stdafx.h"
```

15. stdafx.h 源代码

```
// stdafx.h : include file for standard system include files,
//   or project specific include files that are used frequently, but
//        are changed infrequently
//

#if !defined(AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_)
#define AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000

#define VC_EXTRALEAN        // Exclude rarely-used stuff from Windows headers

#include <afxwin.h>              // MFC core and standard components
#include <afxext.h>              // MFC extensions
#include <afxdisp.h>             // MFC Automation classes
#include <afxdtctl.h>        // MFC support for Internet Explorer 4 Common Controls
#ifndef _AFX_NO_AFXCMN_SUPPORT
#include <afxcmn.h>              // MFC support for Windows Common Controls
#endif // _AFX_NO_AFXCMN_SUPPORT


//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif
// !defined(AFX_STDAFX_H__952C126A_D524_4D63_AB65_EBE6A3F4026B__INCLUDED_)
```

16. ch1_1.rc 源代码

```
//Microsoft Developer Studio generated resource script.
//
```

```
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

/////////////////////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////////////////
// Chinese (P.R.C.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)
#ifdef _WIN32
LANGUAGE LANG_CHINESE, SUBLANG_CHINESE_SIMPLIFIED
#pragma code_page(936)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "#define _AFX_NO_SPLITTER_RESOURCES\r\n"
    "#define _AFX_NO_OLE_RESOURCES\r\n"
    "#define _AFX_NO_TRACKER_RESOURCES\r\n"
    "#define _AFX_NO_PROPERTY_RESOURCES\r\n"
    "\r\n"
    "#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)\r\n"
    "#ifdef _WIN32\r\n"
    "LANGUAGE 4, 2\r\n"
    "#pragma code_page(936)\r\n"
    "#endif //_WIN32\r\n"
    "#include ""res\\ch1_1.rc2""    // non-Microsoft Visual C++ edited resources\r\n"
```

```
    "#include ""l.chs\\afxres.rc""            // Standard components\r\n"
    "#include ""l.chs\\afxprint.rc""          // printing/print preview resources\r\n"
    "#endif\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDR_MAINFRAME           ICON    DISCARDABLE     "res\\ch1_1.ico"
IDR_CH1_1TYPE           ICON    DISCARDABLE     "res\\ch1_1Doc.ico"

/////////////////////////////////////////////////////////////////////////////
//
// Bitmap
//

IDR_MAINFRAME           BITMAP  MOVEABLE PURE   "res\\Toolbar.bmp"

/////////////////////////////////////////////////////////////////////////////
//
// Toolbar
//

IDR_MAINFRAME TOOLBAR DISCARDABLE  16, 15
BEGIN
    BUTTON      ID_FILE_OPEN
    BUTTON      ID_FILE_SAVE
    SEPARATOR
    BUTTON      ID_EDIT_CUT
    BUTTON      ID_EDIT_COPY
    BUTTON      ID_EDIT_PASTE
    SEPARATOR
    BUTTON      ID_FILE_PRINT
    SEPARATOR
    BUTTON      ID_APP_ABOUT
END


/////////////////////////////////////////////////////////////////////////////
//
// Menu
//
```

```
IDR_MAINFRAME MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "文件(&F)"
    BEGIN
        MENUITEM "打开(&O)...\tCtrl+O",         ID_FILE_OPEN
        MENUITEM SEPARATOR
        MENUITEM "打印设置(&R)...",              ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "最近文件",                     ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "退出(&X)",                     ID_APP_EXIT
    END
    POPUP "查看(&V)"
    BEGIN
        MENUITEM "工具栏(&T)",                   ID_VIEW_TOOLBAR
        MENUITEM "状态栏(&S)",                   ID_VIEW_STATUS_BAR
    END
    POPUP "帮助(&H)"
    BEGIN
        MENUITEM "关于 ch1_1(&A)...",            ID_APP_ABOUT
    END
END

IDR_CH1_1TYPE MENU PRELOAD DISCARDABLE
BEGIN
    POPUP "文件(&F)"
    BEGIN
        MENUITEM "打开(&O)...\tCtrl+O",         ID_FILE_OPEN
        MENUITEM "关闭(&C)",                     ID_FILE_CLOSE
        MENUITEM "保存(&S)\tCtrl+S",             ID_FILE_SAVE
        MENUITEM "另存为(&A)...",                ID_FILE_SAVE_AS
        MENUITEM "重新加载(&R)\tCtrl+R",         ID_FILE_REOPEN
        MENUITEM SEPARATOR
        MENUITEM "打印(&P)...\tCtrl+P",          ID_FILE_PRINT
        MENUITEM "打印预览(&V)",                 ID_FILE_PRINT_PREVIEW
        MENUITEM "打印设置(&R)...",              ID_FILE_PRINT_SETUP
        MENUITEM SEPARATOR
        MENUITEM "最近文件",                     ID_FILE_MRU_FILE1, GRAYED
        MENUITEM SEPARATOR
        MENUITEM "退出(&X)",                     ID_APP_EXIT
    END
    POPUP "编辑(&E)"
    BEGIN
        MENUITEM "撤消(&U)\tCtrl+Z",             ID_EDIT_UNDO
        MENUITEM SEPARATOR
        MENUITEM "剪切(&T)\tCtrl+X",             ID_EDIT_CUT
        MENUITEM "复制(&C)\tCtrl+C",             ID_EDIT_COPY
        MENUITEM "粘贴(&P)\tCtrl+V",             ID_EDIT_PASTE
    END
    POPUP "查看(&V)"
```

```
    BEGIN
        MENUITEM "工具栏(&T)",              ID_VIEW_TOOLBAR
        MENUITEM "状态栏(&S)",              ID_VIEW_STATUS_BAR
    END
    POPUP "窗口(&W)"
    BEGIN
        MENUITEM "新建窗口(&N)",             ID_WINDOW_NEW
        MENUITEM "层叠(&C)",                ID_WINDOW_CASCADE
        MENUITEM "平铺(&T)",                ID_WINDOW_TILE_HORZ
        MENUITEM "排列图标(&A)",             ID_WINDOW_ARRANGE
    END
    POPUP "帮助(&H)"
    BEGIN
        MENUITEM "关于 ch1_1(&A)...",        ID_APP_ABOUT
    END
END


/////////////////////////////////////////////////////////////////////////////
//
// Accelerator
//

IDR_MAINFRAME ACCELERATORS PRELOAD MOVEABLE PURE
BEGIN
    "N",            ID_FILE_NEW,            VIRTKEY, CONTROL
    "O",            ID_FILE_OPEN,           VIRTKEY, CONTROL
    "S",            ID_FILE_SAVE,           VIRTKEY, CONTROL
    "P",            ID_FILE_PRINT,          VIRTKEY, CONTROL
    "Z",            ID_EDIT_UNDO,           VIRTKEY, CONTROL
    "X",            ID_EDIT_CUT,            VIRTKEY, CONTROL
    "C",            ID_EDIT_COPY,           VIRTKEY, CONTROL
    "V",            ID_EDIT_PASTE,          VIRTKEY, CONTROL
    VK_BACK,        ID_EDIT_UNDO,           VIRTKEY, ALT
    VK_DELETE,      ID_EDIT_CUT,            VIRTKEY, SHIFT
    VK_INSERT,      ID_EDIT_COPY,           VIRTKEY, CONTROL
    VK_INSERT,      ID_EDIT_PASTE,          VIRTKEY, SHIFT
    VK_F6,          ID_NEXT_PANE,           VIRTKEY
    VK_F6,          ID_PREV_PANE,           VIRTKEY, SHIFT
END


/////////////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 235, 55
STYLE DS_MODALFRAME | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "关于 ch1_1"
```

```
FONT 9, "宋体"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,20,20
    LTEXT           "ch1_1 1.0 版",IDC_STATIC,40,10,119,8,SS_NOPREFIX
    LTEXT           "版权所有 (C) 2000",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "确定",IDOK,178,7,50,14,WS_GROUP
END

#ifndef _MAC
/////////////////////////////////////////////////////////////////////////////
//
// Version
//

VS_VERSION_INFO VERSIONINFO
 FILEVERSION 1,0,0,1
 PRODUCTVERSION 1,0,0,1
 FILEFLAGSMASK 0x3fL
#ifdef _DEBUG
 FILEFLAGS 0x1L
#else
 FILEFLAGS 0x0L
#endif
 FILEOS 0x4L
 FILETYPE 0x1L
 FILESUBTYPE 0x0L
BEGIN
    BLOCK "StringFileInfo"
    BEGIN
        BLOCK "080404B0"
        BEGIN
            VALUE "CompanyName", "\0"
            VALUE "FileDescription", "ch1_1 Microsoft 基础类应用程序\0"
            VALUE "FileVersion", "1, 0, 0, 1\0"
            VALUE "InternalName", "ch1_1\0"
            VALUE "LegalCopyright", "版权所有 (C) 2000\0"
            VALUE "LegalTrademarks", "\0"
            VALUE "OriginalFilename", "ch1_1.EXE\0"
            VALUE "ProductName", "ch1_1 应用程序\0"
            VALUE "ProductVersion", "1, 0, 0, 1\0"
        END
    END
    BLOCK "VarFileInfo"
    BEGIN
        VALUE "Translation", 0x804, 1200
    END
END

#endif    // !_MAC
```

```
/////////////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_ABOUTBOX, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 228
        TOPMARGIN, 7
        BOTTOMMARGIN, 48
    END

    IDD_DLG_GEOTrans, DIALOG
    BEGIN
        LEFTMARGIN, 4
        RIGHTMARGIN, 118
        TOPMARGIN, 7
        BOTTOMMARGIN, 71
    END

    IDD_DLG_GEOZoom, DIALOG
    BEGIN
        LEFTMARGIN, 4
        RIGHTMARGIN, 115
        TOPMARGIN, 7
        BOTTOMMARGIN, 69
    END
END
#endif    // APSTUDIO_INVOKED


/////////////////////////////////////////////////////////////////////////////
//
// String Table
//

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    IDR_MAINFRAME           "Visual C++数字图像编程基础"
    IDR_CH1_1TYPE           "\nBMP图像文件\nBMP图像文件\nBMP图像文件
(*.BMP)\n.BMP\nCh11.Document\nCh1_1 Document"
END

STRINGTABLE PRELOAD DISCARDABLE
BEGIN
    AFX_IDS_APP_TITLE       "Visual C++数字图像编程基础"
```

```
    AFX_IDS_IDLEMESSAGE       "就绪"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_INDICATOR_EXT          "扩展名"
    ID_INDICATOR_CAPS         "大写"
    ID_INDICATOR_NUM          "数字"
    ID_INDICATOR_SCRL         "滚动"
    ID_INDICATOR_OVR          "改写"
    ID_INDICATOR_REC          "记录"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_NEW               "建立新文档\n新建"
    ID_FILE_OPEN              "打开一个现有文档\n打开"
    ID_FILE_CLOSE             "关闭活动文档\n关闭"
    ID_FILE_SAVE              "保存活动文档\n保存"
    ID_FILE_SAVE_AS           "将活动文档以一个新文件名保存\n另存为"
    ID_FILE_PAGE_SETUP        "改变打印选项\n页面设置"
    ID_FILE_PRINT_SETUP       "改变打印机及打印选项\n打印设置"
    ID_FILE_PRINT             "打印活动文档\n打印"
    ID_FILE_PRINT_PREVIEW     "显示整页\n打印预览"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_APP_ABOUT              "显示程序信息，版本号和版权\n关于"
    ID_APP_EXIT               "退出应用程序，提示保存文档\n退出"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_MRU_FILE1         "打开该文档"
    ID_FILE_MRU_FILE2         "打开该文档"
    ID_FILE_MRU_FILE3         "打开该文档"
    ID_FILE_MRU_FILE4         "打开该文档"
    ID_FILE_MRU_FILE5         "打开该文档"
    ID_FILE_MRU_FILE6         "打开该文档"
    ID_FILE_MRU_FILE7         "打开该文档"
    ID_FILE_MRU_FILE8         "打开该文档"
    ID_FILE_MRU_FILE9         "打开该文档"
    ID_FILE_MRU_FILE10        "打开该文档"
    ID_FILE_MRU_FILE11        "打开该文档"
    ID_FILE_MRU_FILE12        "打开该文档"
    ID_FILE_MRU_FILE13        "打开该文档"
    ID_FILE_MRU_FILE14        "打开该文档"
    ID_FILE_MRU_FILE15        "打开该文档"
    ID_FILE_MRU_FILE16        "打开该文档"
END
```

```
STRINGTABLE DISCARDABLE
BEGIN
    ID_NEXT_PANE                "切换到下一个窗格\n下一窗格"
    ID_PREV_PANE                "切换回前一个窗格\n前一窗格"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_WINDOW_NEW               "为活动文档打开另一个窗口\n新建窗口"
    ID_WINDOW_ARRANGE           "将图标排列在窗口底部\n排列图标"
    ID_WINDOW_CASCADE           "排列窗口成相互重叠\n层叠窗口"
    ID_WINDOW_TILE_HORZ         "排列窗口成互不重叠\n平铺窗口"
    ID_WINDOW_TILE_VERT         "排列窗口成互不重叠\n平铺窗口"
    ID_WINDOW_SPLIT             "将活动的窗口分隔成窗格\n分隔"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_EDIT_CLEAR               "删除被选对象\n删除"
    ID_EDIT_CLEAR_ALL           "全部删除\n全部删除"
    ID_EDIT_COPY                "复制被选对象并将其置于剪贴板上\n复制"
    ID_EDIT_CUT                 "剪切被选对象并将其置于剪贴板上\n剪切"
    ID_EDIT_FIND                "查找指定的正文\n查找"
    ID_EDIT_PASTE               "插入剪贴板内容\n粘贴"
    ID_EDIT_REPEAT              "重复上一步操作\n重复"
    ID_EDIT_REPLACE             "用不同的正文替换指定的正文\n替换"
    ID_EDIT_SELECT_ALL          "选择整个文档\n选择全部"
    ID_EDIT_UNDO                "撤消最后一步操作\n撤消"
    ID_EDIT_REDO                "重新执行先前已撤消的操作\n重新执行"
END

STRINGTABLE DISCARDABLE
BEGIN
    ID_VIEW_TOOLBAR             "显示或隐藏工具栏\n显隐工具栏"
    ID_VIEW_STATUS_BAR          "显示或隐藏状态栏\n显隐状态栏"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_SCSIZE              "改变窗口大小"
    AFX_IDS_SCMOVE              "改变窗口位置"
    AFX_IDS_SCMINIMIZE          "将窗口缩小成图标"
    AFX_IDS_SCMAXIMIZE          "把窗口放大到最大尺寸"
    AFX_IDS_SCNEXTWINDOW        "切换到下一个文档窗口"
    AFX_IDS_SCPREVWINDOW        "切换到先前的文档窗口"
    AFX_IDS_SCCLOSE             "关闭活动窗口并提示保存所有文档"
END

STRINGTABLE DISCARDABLE
```

```
BEGIN
    AFX_IDS_SCRESTORE          "把窗口恢复到正常大小"
    AFX_IDS_SCTASKLIST         "激活任务表"
    AFX_IDS_MDICHILD           "激活该窗口"
END

STRINGTABLE DISCARDABLE
BEGIN
    AFX_IDS_PREVIEW_CLOSE     "关闭打印预览模式\n取消预阅"
END

#endif    // Chinese (P.R.C.) resources
/////////////////////////////////////////////////////////////////////////////



#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
#define _AFX_NO_SPLITTER_RESOURCES
#define _AFX_NO_OLE_RESOURCES
#define _AFX_NO_TRACKER_RESOURCES
#define _AFX_NO_PROPERTY_RESOURCES

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_CHS)
#ifdef _WIN32
LANGUAGE 4, 2
#pragma code_page(936)
#endif //_WIN32
#include "res\ch1_1.rc2"  // non-Microsoft Visual C++ edited resources
#include "l.chs\afxres.rc"          // Standard components
#include "l.chs\afxprint.rc"        // printing/print preview resources
#endif

/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

注意，上面的程序用到了上小节构造的 DIB 函数库。编译该程序时需要加入 dibapi.cpp 和 dibapi.h 两个文件。

第三章　图像的点运算

在上一章中我们介绍了位图和 DIB 的概念以及如何读写 BMP 图像文件。在本章中我们将介绍图像的点运算。

点运算（Point Operation）是一种既简单又重要的技术，它能让用户改变图像数据占据的灰度范围。一幅输入图像经过点运算后将产生一幅新的输出图像，由输入像素点的灰度值决定相应的输出像素点的灰度值。点运算与后面将要介绍的局部运算的差别在于：后者每个输出像素的灰度值由对应输入像素的一个邻域内几个像素的灰度值决定。因此，点运算不可能改变图像内的空间关系。

点运算可以按照预定的方式改变一幅图像的灰度直方图。除了灰度级的改变是根据某种特定的灰度变换函数进行之外，点运算可以看作是“从像素到像素”的复制操作。如果输入图像为 $A(x, y)$，输出图像为 $B(x, y)$，则点运算可表示为：

$$B(x, y) = f[A(x, y)]$$

其中函数 $f(D)$ 被称为灰度变换（Gray Scale Transformation, GST）函数，它描述了输入灰度值和输出灰度值之间的转换关系。一旦灰度变换函数确定，该点运算就完全被确定下来了。

点运算有时又被称为对比度增强、对比度拉伸或灰度变换，它是图像数字化软件和图像显示软件的重要组成部分。本章将介绍图像灰度的线性变换、窗运算、灰度拉伸和灰度均衡等几种常见的点运算。在介绍每种点运算时，先介绍该种运算的理论基础，接下来介绍如何在 Visual C++中编程实现该运算。

3.1　灰度直方图

灰度直方图是数字图像处理中一个最简单、最有用的工具，它描述了一幅图像的灰度级内容。任何一幅图像的直方图都包括了可观的信息，某些类型的图像还可由其直方图完全描述。

3.1.1　灰度直方图的定义

灰度直方图是灰度值的函数，描述的是图像中具有该灰度值的像素的个数，其横坐标表示像素的灰度级别，纵坐标是该灰度出现的频率（像素的个数）。图 3－1 的灰度直方图如图 3－2 所示。

灰度直方图也有另一种方式的定义：假设有一幅由函数 $D(x, y)$ 所定义的连续图像，它平滑地从中心的高灰度级变化到边沿的低灰度级。我们可以选择某一灰度级 D_1，然后定义一条轮廓线，该轮廓线连接了图像上所有具有灰度级 D_1 的点。所得到的轮廓线形成了包围灰度级大于等于 D_1 的区域的封闭曲线。如图 3－3 所示，图像中有一条灰度级为 D_1 的轮廓线，在更

高的灰度级 D_2 处还有第二条轮廓线。设 A_1 是第一条轮廓线所包围区域的面积，A_2 是第二条轮廓线所包围的区域的面积。



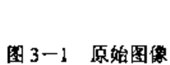


图 3—1 原始图像

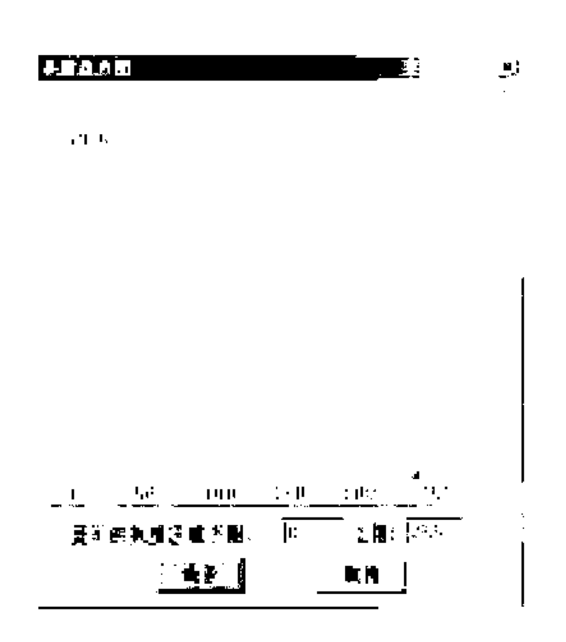


图 3—2 图像灰度直方图

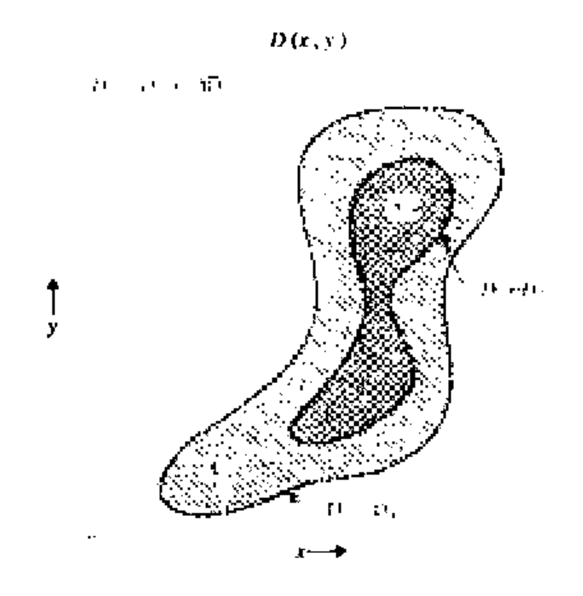


图 3—3 图像的灰度轮廓线

将一幅连续图像中被具有灰度级 D 的所有轮廓线所包围的面积称为该图像的阈值面积函数 $A(D)$。直方图可定义为：

$$H(D)=\lim_{\Delta D\to 0}\frac{A(D)-A(D+\Delta D)}{\Delta D}=-\frac{d}{dD}A(D)$$

出上式可以得出结论：一幅连续图像的直方图是其面积函数的导数的负值。负号的出现是由于随着 D 的增加 A(D)在减小。如果将图像看成是一个二维的随机变量，则面积函数相当于其累积分布函数，而灰度直方图相当于其概率密度函数。

对于离散函数，我们固定 ΔD 为 1，则上式变为：

$$H(D)=A(D)-A(D+1)$$

3.1.2 编程绘制灰度直方图

下面我们来完善上章开发的数字图像处理程序。首先在程序中添加一个对话框，该对话框对应的类为 CdlgIntensity。要在对话框中绘制出图像的灰度直方图，首先必须得到图像像素的指针（保存在类成员变量 m_lpDIBBits 中）和图像的高度、宽度信息（保存在类成员变量 m_ lHeight 和 m_lWidth 中）。有了这些信息，就可以计算出各个灰度的像素数（保存在类成员变量 m_ lCount[256]数组中）。在显示灰度直方图时，还存在这样一个问题：当某个灰度的计数远远大于其他灰度的计数时，灰度直方图往往看不清图像真正的灰度分布。例如图 3－1 中，灰度为白色（灰度值为 255）的像素很多，达到了 69 768 个，而其他灰度的计数相对太少，因此在图 3－1 的灰度直方图 3－2 中只能看到灰度值为 255 附近的几个灰度的分布，看不到某个区间的具体灰度分布。但在绘制灰度直方图的程序中，允许用户指定查看某个区间的灰度分布。在类 CdlgIntensity 中，我们定义了两个成员变量来保存用户的设置：m_iLowGray 保存用户指定查看区间的起始灰度；m_iUpGray 保存查看区间的上限。图 3－4 就是图 3－1 中图像灰度在 6~254 之间的灰度直方图。这样就可以很清楚地看到各个灰度的分布了。

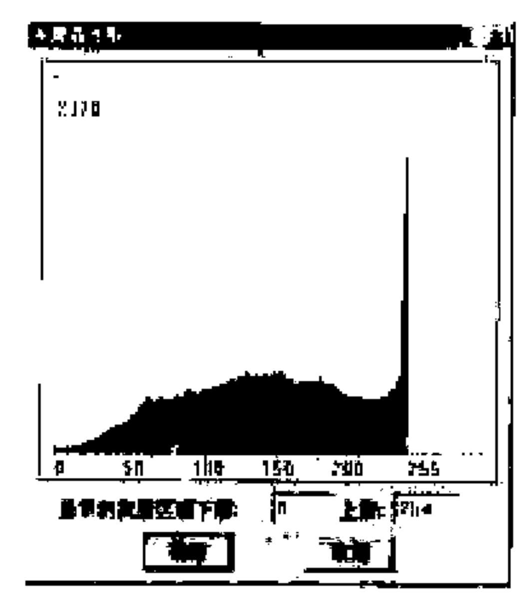


图 3－4　图像的灰度直方图（区间显示）

下面给出了该对话框的完整代码。

1. 对话框头文件（DlgIntensity.h）

```
#if !defined(AFX_DLGINTENSITY_H__504BB8CE_7CF6_4D13_B5B7_DCC1BC84FEA5__INCLUDED_)
#define AFX_DLGINTENSITY_H__504BB8CE_7CF6_4D13_B5B7_DCC1BC84FEA5__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgIntensity.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgIntensity dialog

class CDlgIntensity : public CDialog
{
// Construction
public:

    // 当前鼠标拖动状态，0表示未拖动，1表示正在拖动下限，2表示正在拖动上限。
    int      m_iIsDraging;

    // 相应鼠标事件的矩形区域
    CRect    m_MouseRect;

    // DIB的高度
    LONG     m_lHeight;

    // DIB的宽度
    LONG     m_lWidth;

    // 指向当前DIB像素的指针
    char *   m_lpDIBBits;

    // 各个灰度值的计数
    LONG     m_lCount[256];

    CDlgIntensity(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgIntensity)
    enum { IDD = IDD_DLG_Intensity };

    // 显示灰度区间的下限
    int      m_iLowGray;

    // 显示灰度区间的上限
    int      m_iUpGray;
    //}}AFX_DATA
```

```
// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgIntensity)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgIntensity)
    afx_msg void OnPaint();
    afx_msg void OnKillfocusEDITLowGray();
    afx_msg void OnKillfocusEDITUpGray();
    virtual void OnOK();
    virtual BOOL OnInitDialog();
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_DLGINTENSITY_H__504BB8CE_7CF6_4D13_B5B7_DCC1BC84FEA5__INCLUDED_)
```

2. 对话框程序（DlgIntensity.cpp）

```
// DlgIntensity.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgIntensity.h"
#include "DIBAPI.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgIntensity dialog


CDlgIntensity::CDlgIntensity(CWnd* pParent /*=NULL*/)
```

```
    : CDialog(CDlgIntensity::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgIntensity)
    m_iLowGray = 0;
    m_iUpGray = 0;
    //}}AFX_DATA_INIT
}


void CDlgIntensity::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgIntensity)
    DDX_Text(pDX, IDC_EDIT_LowGray, m_iLowGray);
    DDV_MinMaxInt(pDX, m_iLowGray, 0, 255);
    DDX_Text(pDX, IDC_EDIT_UpGray, m_iUpGray);
    DDV_MinMaxInt(pDX, m_iUpGray, 0, 255);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgIntensity, CDialog)
    //{{AFX_MSG_MAP(CDlgIntensity)
    ON_WM_PAINT()
    ON_EN_KILLFOCUS(IDC_EDIT_LowGray, OnKillfocusEDITLowGray)
    ON_EN_KILLFOCUS(IDC_EDIT_UpGray, OnKillfocusEDITUpGray)
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgIntensity message handlers

BOOL CDlgIntensity::OnInitDialog()
{
    // 指向原图像像素的指针
    unsigned char * lpSrc;

    // 循环变量
    LONG i;
    LONG j;

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();

    // 获取绘制直方图的标签
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 计算接受鼠标事件的有效区域
```

```
    pWnd->GetClientRect(m_MouseRect);
    pWnd->ClientToScreen(&m_MouseRect);

    CRect rect;
    GetClientRect(rect);
    ClientToScreen(&rect);

    m_MouseRect.top -= rect.top;
    m_MouseRect.left -= rect.left;

    // 设置接受鼠标事件的有效区域
    m_MouseRect.top += 25;
    m_MouseRect.left += 10;
    m_MouseRect.bottom = m_MouseRect.top + 255;
    m_MouseRect.right = m_MouseRect.left + 256;

    // 重置计数为0
    for (i = 0; i < 256; i ++)
    {
        // 清零
        m_lCount[i] = 0;
    }

    // 图像每行的字节数
    LONG lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(m_lWidth * 8);

    // 计算各个灰度值的计数
    for (i = 0; i < m_lHeight; i ++)
    {
        for (j = 0; j < m_lWidth; j ++)
        {
            lpSrc = (unsigned char *)m_lpDIBBits + lLineBytes * i + j;

            // 计数加1
            m_lCount[*(lpSrc)]++;
        }
    }

    // 初始化拖动状态
    m_iIsDraging = 0;

    // 返回TRUE
    return TRUE;
}

void CDlgIntensity::OnKillfocusEDITLowGray()
{
    // 保存变动（控件中数值保存到成员变量中）
```

```
    UpdateData(TRUE);

    // 判断是否下限超过上限
    if (m_iLowGray > m_iUpGray)
    {
        // 互换
        int iTemp = m_iLowGray;
        m_iLowGray = m_iUpGray;
        m_iUpGray = iTemp;

        // 更新（成员变量中数值保存到控件中）
        UpdateData(FALSE);
    }

    // 重绘直方图
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgIntensity::OnKillfocusEDITUpGray()
{
    // 保存变动
    UpdateData(TRUE);

    // 判断下限是否超过上限
    if (m_iLowGray > m_iUpGray)
    {
        // 互换
        int iTemp = m_iLowGray;
        m_iLowGray = m_iUpGray;
        m_iUpGray = iTemp;

        // 更新
        UpdateData(FALSE);
    }

    // 重绘直方图
    InvalidateRect(m_MouseRect, TRUE);

}

void CDlgIntensity::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当用户单击鼠标左键开始拖动

    // 判断是否在接受鼠标事件的有效区域中
    if(m_MouseRect.PtInRect(point))
    {
        if (point.x == (m_MouseRect.left + m_iLowGray))
        {

            // 设置拖动状态1，拖动下限
```

```
            m_iIsDraging = 1;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
        else if (point.x == (m_MouseRect.left + m_iUpGray))
        {

            // 设置拖动状态为2，拖动上限
            m_iIsDraging = 2;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
    }

    // 默认单击鼠标左键处理事件
    CDialog::OnLButtonDown(nFlags, point);
}

void CDlgIntensity::OnMouseMove(UINT nFlags, CPoint point)
{

    // 判断是否在接受鼠标事件的有效区域中
    if(m_MouseRect.PtInRect(point))
    {
        // 判断是否正在拖动
        if (m_iIsDraging != 0)
        {
            // 判断正在拖动上限还是下限
            if (m_iIsDraging == 1)
            {
                // 判断是否下限<上限
                if (point.x - m_MouseRect.left < m_iUpGray)
                {
                    // 更改下限
                    m_iLowGray = point.x - m_MouseRect.left;
                }
                else
                {
                    // 下限拖过上限，设置为上限-1
                    m_iLowGray = m_iUpGray - 1;

                    // 重设鼠标位置
                    point.x = m_MouseRect.left + m_iUpGray - 1;
                }
            }
            else
            {
                // 正在拖动上限
```

```
                // 判断是否上限>下限
                if (point.x - m_MouseRect.left > m_iLowGray)
                {
                    // 更改下限
                    m_iUpGray = point.x - m_MouseRect.left;
                }
                else
                {
                    // 下限拖过上限，设置为下限+1
                    m_iUpGray = m_iLowGray + 1;

                    // 重设鼠标位置
                    point.x = m_MouseRect.left + m_iLowGray + 1;
                }
            }

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));

            // 更新
            UpdateData(FALSE);

            // 重绘直方图
            InvalidateRect(m_MouseRect, TRUE);
        }
        else if (point.x == (m_MouseRect.left + m_iLowGray)
                || point.x == (m_MouseRect.left + m_iUpGray))
        {
            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
    }

    // 默认鼠标移动处理事件
    CDialog::OnMouseMove(nFlags, point);
}

void CDlgIntensity::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 当用户释放鼠标左键停止拖动
    if (m_iIsDraging != 0)
    {
        // 重置拖动状态
        m_iIsDraging = 0;
    }

    // 默认释放鼠标左键处理事件
    CDialog::OnLButtonUp(nFlags, point);
}

void CDlgIntensity::OnPaint()
```

```
{
    // 字符串
    CString str;

    // 循环变量
    LONG i;

    // 最大计数
    LONG lMaxCount;

    // 设备上下文
    CPaintDC dc(this);

    // 获取绘制坐标的文本框
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 指针
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();

    pDC->Rectangle(0,0,330,300);

    // 创建画笔对象
    CPen* pPenRed = new CPen;

    // 红色画笔
    pPenRed->CreatePen(PS_SOLID,1,RGB(255,0,0));

    // 创建画笔对象
    CPen* pPenBlue = new CPen;

    // 蓝色画笔
    pPenBlue->CreatePen(PS_SOLID,1,RGB(0,0, 255));

    // 创建画笔对象
    CPen* pPenGreen = new CPen;

    // 绿色画笔
    pPenGreen->CreatePen(PS_DOT,1,RGB(0,255,0));

    // 选中当前红色画笔，并保存以前的画笔
    CGdiObject* pOldPen = pDC->SelectObject(pPenRed);

    // 绘制坐标轴
    pDC->MoveTo(10,10);

    // 垂直轴
    pDC->LineTo(10,280);

    // 水平轴
```

```
pDC->LineTo(320,280);

// 写X轴刻度值
str.Format("0");
pDC->TextOut(10, 283, str);
str.Format("50");
pDC->TextOut(60, 283, str);
str.Format("100");
pDC->TextOut(110, 283, str);
str.Format("150");
pDC->TextOut(160, 283, str);
str.Format("200");
pDC->TextOut(210, 283, str);
str.Format("255");
pDC->TextOut(265, 283, str);

// 绘制X轴刻度
for (i = 0; i < 256; i += 5)
{
    if ((i & 1) == 0)
    {
        // 10的倍数
        pDC->MoveTo(i + 10, 280);
        pDC->LineTo(i + 10, 284);
    }
    else
    {
        // 10的倍数
        pDC->MoveTo(i + 10, 280);
        pDC->LineTo(i + 10, 282);
    }
}

// 绘制X轴箭头
pDC->MoveTo(315,275);
pDC->LineTo(320,280);
pDC->LineTo(315,285);

// 绘制X轴箭头
pDC->MoveTo(10,10);
pDC->LineTo(5,15);
pDC->MoveTo(10,10);
pDC->LineTo(15,15);

// 计算最大计数值
for (i = m_iLowGray; i <= m_iUpGray; i ++)
{
    // 判断是否大于当前最大值
    if (m_lCount[i] > lMaxCount)
    {
        // 更新最大值
```

```
            lMaxCount = m_lCount[i];
        }
    }

    // 输出最大计数值
    pDC->MoveTo(10, 25);
    pDC->LineTo(14, 25);
    str.Format("%d", lMaxCount);
    pDC->TextOut(11, 26, str);

    // 更改成绿色画笔
    pDC->SelectObject(pPenGreen);

    // 绘制窗口上下限
    pDC->MoveTo(m_iLowGray + 10, 25);
    pDC->LineTo(m_iLowGray + 10, 280);

    pDC->MoveTo(m_iUpGray + 10, 25);
    pDC->LineTo(m_iUpGray + 10, 280);

    // 更改成蓝色画笔
    pDC->SelectObject(pPenBlue);


    // 判断是否有计数
    if (lMaxCount > 0)
    {
        // 绘制直方图
        for (i = m_iLowGray; i <= m_iUpGray; i ++)
        {
            pDC->MoveTo(i + 10, 280);
            pDC->LineTo(i + 10, 281 - (int) (m_lCount[i] * 256 / lMaxCount));
        }
    }

    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);

    // 删除新的画笔
    delete pPenRed;
    delete pPenBlue;
    delete pPenGreen;

}

void CDlgIntensity::OnOK()
{

    // 判断是否下限超过上限
    if (m_iLowGray > m_iUpGray)
    {
```

```
        // 互换
        int iTemp = m_iLowGray;
        m_iLowGray = m_iUpGray;
        m_iUpGray = iTemp;
    }

    // 返回
    CDialog::OnOK();
}
```

接下来在“查看”菜单中添加一个名为直方图的菜单项。如图 3—5 所示。

图 3—5 添加直方图菜单项

在类 CCh1_1View 中添加该菜单事件的处理程序：

```
void CCh1_1View::OnViewIntensity()
{
    // 查看当前图像灰度直方图

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持查看256色位图灰度直方图！", "系统提示",
                MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
```

```
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 创建对话框
    CDlgIntensity dlgPara;

    // 初始化变量值
    dlgPara.m_lpDIBBits = lpDIBBits;
    dlgPara.m_lWidth = ::DIBWidth(lpDIB);
    dlgPara.m_lHeight = ::DIBHeight(lpDIB);
    dlgPara.m_iLowGray = 0;
    dlgPara.m_iUpGray = 255;

    // 显示对话框，提示用户设定平移量
    if (dlgPara.DoModal() != IDOK)
    {
        // 返回
        return;
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

3.2 灰度的线性变换

灰度的线性变换是点运算中最简单的运算之一。下面介绍一下灰度的线性变换理论基础：

3.2.1 理论基础

灰度的线性变换就是将图像中所有的点的灰度按照线性灰度变换函数进行变换，该线性灰度变换函数 $f(x)$ 是一个一维线性函数：

$$f(x) = fA \cdot x + fB$$

灰度变换方程为：

$$D_B = f(D_A) = fA \cdot D_A + fB$$

式中参数 fA 为线性函数的斜率，fB 为线性函数的在 y 轴的截距，D_A 表示输入图像的灰

度，D_B表示输出图像的灰度。当$fA>1$时，输出图像的对比度将增大；当$fA<1$时，输出图像的对比度将减小；当$fA=1$且$fB\neq 0$时，操作仅使所有像素的灰度值上移或下移，其效果是使整个图像更暗或更亮；如果$fA<0$，暗区域将变亮，亮区域将变暗，点运算完成了图像求补运算。特殊情况下，当$fA=1$，$fB=0$时，输出图像和输入图像相同；当$fA=-1$，$fB=255$时，输出图像的灰度正好反转。反转图3－1所示的图像的结果如图3－6所示。



图3－6 反色后的图像

3.2.2 Visual C++编程实现

有了上面的理论基础，就可以容易地用Visual C++来实现图像灰度的线性变换。下面我们开始构造自己的图像点运算函数库。首先来完成图像灰度的线性变换。图像灰度的线性变换操作不需要改变DIB的调色板和文件头，只要把指向DIB像素起始位置的指针和DIB高度、宽度信息传递给子函数就可以完成。下面代码中定义的LinerTrans ()函数就是用来实现图像灰度的线性变换的。

```
/*************************************************************************
 *
 * 函数名称:
 *   LinerTrans()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   FLOAT fA           - 线性变换的斜率
 *   FLOAT fB           - 线性变换的截距
 *
 * 返回值:
```

```
 *   BOOL                 - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行灰度的线性变换操作。
 *
 ************************************************************************/

BOOL WINAPI LinerTrans(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, FLOAT fA, FLOAT fB)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 中间变量
    FLOAT   fTemp;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 线性变换
            fTemp = fA * (*lpSrc) + fB;

            // 判断是否超出范围
            if (fTemp > 255)
            {
                // 直接赋值为255
                *lpSrc = 255;
            }
            else if (fTemp < 0)
            {
                // 直接赋值为0
                *lpSrc = 0;
            }
            else
            {
```

```
                // 四舍五入
                *lpSrc = (unsigned char) (fTemp + 0.5);
            }
        }
    }

    // 返回
    return TRUE;

}
```

接下来添加一个名为“点运算”的菜单，并添加我们将要介绍的各种点运算菜单项。如图 3－7 所示。

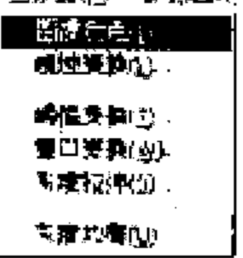


图 3－7 添加点运算菜单

在类 CCh1_1View 中添加图像反色和线性变换菜单项的事件处理程序：

```
void CCh1_1View::OnPointInvert()
{
    // 图像反色

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 线性变换的斜率
    FLOAT fA;

    // 线性变换的截距
    FLOAT fB;

    // 反色操作的线性变换的方程是-x + 255
    fA = -1.0;
    fB = 255.0;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());
```

```
    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的反色，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的反色！", "系统提示",
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 调用LinerTrans()函数反色
    ::LinerTrans(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), fA, fB);

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}

void CCh1_1View::OnPointLiner()
{
    // 线性变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;
```

```
// 参数对话框
CDlgLinerPara dlgPara;

// 线性变换的斜率
FLOAT fA;

// 线性变换的截距
FLOAT fB;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的线性变换，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的线性变换！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 初始化变量值
dlgPara.m_fA = 2.0;
dlgPara.m_fB = -128.0;

// 显示对话框，提示用户设定平移量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户设定的平移量
fA = dlgPara.m_fA;
fB = dlgPara.m_fB;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 调用LinerTrans()函数进行线性变换
```

```
    LinerTrans(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), fA, fB);

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

上面代码中 CdlgLinerPara 是一个新建的对话框类，该对话框主要是用来让用户设置线性变换的参数：斜率 m_fA 和截距 m_fB。该对话框的完整代码如下：

1. 对话框头文件 DlgLinerPara.h

```
#if !defined(AFX_DLGLINERPARA_H__CA65EBE3_E61B_4E4B_9C27_716F1FD13500__INCLUDED_)
#define AFX_DLGLINERPARA_H__CA65EBE3_E61B_4E4B_9C27_716F1FD13500__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgLinerPara.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgLinerPara dialog

class CDlgLinerPara : public CDialog
{
// Construction
public:

    // 标识是否已经绘制橡皮筋线
    BOOL    m_bDrawed;

    // 保存鼠标左键单击时的位置
    CPoint  m_p1;

    // 保存鼠标拖动时的位置
    CPoint  m_p2;

    // 当前鼠标拖动状态，TRUE表示正在拖动。
    BOOL    m_bIsDraging;

    // 相应鼠标事件的矩形区域
    CRect   m_MouseRect;
```

```
    CDlgLinerPara(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgLinerPara)
    enum { IDD = IDD_DLG_LinerPara };

    // 线性变换的斜率
    float   m_fA;

    // 线性变换的截距
    float   m_fB;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgLinerPara)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgLinerPara)
    afx_msg void OnPaint();
    afx_msg void OnKillfocusEditA();
    afx_msg void OnKillfocusEditB();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    virtual BOOL OnInitDialog();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_DLGLINERPARA_H__CA65EBE3_E61B_4E4B_9C27_716F1FD13500__INCLUDED_)
```

2. 对话框代码 DlgLinerPara.cpp

```
// DlgLinerPara.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgLinerPara.h"
```

```
#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgLinerPara dialog


CDlgLinerPara::CDlgLinerPara(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgLinerPara::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgLinerPara)
    m_fA = 0.0f;
    m_fB = 0.0f;
    //}}AFX_DATA_INIT
}


void CDlgLinerPara::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgLinerPara)
    DDX_Text(pDX, IDC_EDIT_A, m_fA);
    DDX_Text(pDX, IDC_EDIT_B, m_fB);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgLinerPara, CDialog)
    //{{AFX_MSG_MAP(CDlgLinerPara)
    ON_WM_PAINT()
    ON_EN_KILLFOCUS(IDC_EDIT_A, OnKillfocusEditA)
    ON_EN_KILLFOCUS(IDC_EDIT_B, OnKillfocusEditB)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgLinerPara message handlers

BOOL CDlgLinerPara::OnInitDialog()
{

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();
```

```
    // 获取绘制直方图的标签
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 计算接受鼠标事件的有效区域
    pWnd->GetClientRect(m_MouseRect);
    pWnd->ClientToScreen(&m_MouseRect);

    CRect rect;
    GetClientRect(rect);
    ClientToScreen(&rect);

    m_MouseRect.top -= rect.top;
    m_MouseRect.left -= rect.left;

    // 设置接受鼠标事件的有效区域
    m_MouseRect.top += 25;
    m_MouseRect.left += 10;
    m_MouseRect.bottom = m_MouseRect.top + 255;
    m_MouseRect.right = m_MouseRect.left + 256;

    // 初始化拖动状态
    m_bIsDraging = FALSE;

    // 返回TRUE
    return TRUE;
}

void CDlgLinerPara::OnKillfocusEditA()
{
    // 保存用户设置
    UpdateData(TRUE);

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgLinerPara::OnKillfocusEditB()
{
    // 保存用户设置
    UpdateData(TRUE);

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgLinerPara::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当用户单击鼠标左键开始拖动
    if(m_MouseRect.PtInRect(point))
    {
        // 保存当前鼠标位置
```

```
        m_p1 = point;

        // 转换坐标系
        m_p1.x = m_p1.x - m_MouseRect.left + 10;
        m_p1.y = m_p1.y - m_MouseRect.top + 25;

        // 设置拖动状态
        m_bIsDraging = TRUE;

        // 设置m_bDrawed为FALSE
        m_bDrawed = FALSE;

        // 更改光标
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));

        // 开始跟踪鼠标事件（保证当鼠标移动到窗体外时也可以接收到鼠标释放事件）
        SetCapture();
    }
    // 默认单击鼠标左键处理事件
    CDialog::OnLButtonDown(nFlags, point);
}

void CDlgLinerPara::OnMouseMove(UINT nFlags, CPoint point)
{
    // 判断当前光标是否在绘制区域
    if(m_MouseRect.PtInRect(point))
    {
        // 更改光标
        ::SetCursor(::LoadCursor(NULL, IDC_CROSS));

        // 判断是否正在拖动
        if (m_bIsDraging)
        {
            // 获取绘图的标签
            CWnd* pWnd = GetDlgItem(IDC_COORD);

            // 获取设备上下文
            CDC* pDC = pWnd->GetDC();

            // 设置绘制方式为异或模式
            int nOldDrawMode = pDC->SetROP2(R2_XORPEN);

            // 创建新的画笔
            CPen* pPen = new CPen;
            pPen->CreatePen(PS_DOT,1,RGB(0,0,0));

            // 选中新画笔
            CGdiObject* pOldPen = pDC->SelectObject(pPen);

            // 判断是否已经画过橡皮筋线
            if (m_bDrawed)
```

```
            {
                // 擦去以前的橡皮筋线
                pDC->MoveTo(m_p1);
                pDC->LineTo(m_p2);
            }

            // 保存当前的坐标
            m_p2 = point;

            // 转换坐标系
            m_p2.x = m_p2.x - m_MouseRect.left + 10;
            m_p2.y = m_p2.y - m_MouseRect.top + 25;

            // 绘制一条新橡皮筋线
            pDC->MoveTo(m_p1);
            pDC->LineTo(m_p2);

            // 设置m_bDrawed为TRUE
            m_bDrawed = TRUE;

            // 选回以前的画笔
            pDC->SelectObject(pOldPen);

            // 恢复成以前的绘制模式
            pDC->SetROP2(nOldDrawMode);

            delete pPen;
            ReleaseDC(pDC);
        }
    }
    else
    {
        // 判断是否正在拖动
        if (m_bIsDraging)
        {
            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_NO));
        }
    }

    // 默认鼠标移动处理事件
    CDialog::OnMouseMove(nFlags, point);
}

void CDlgLinerPara::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 当用户释放鼠标左键停止拖动
    if (m_bIsDraging)
    {
        // 判断当前光标是否在绘制区域
        if(m_MouseRect.PtInRect(point))
```

```
{
    // 保存当前鼠标位置
    m_p2 = point;

    // 转换坐标系
    m_p2.x = m_p2.x - m_MouseRect.left + 10;
    m_p2.y = m_p2.y - m_MouseRect.top + 25;

    if ((m_p1 != m_p2) && (m_p1.x != m_p2.x))
    {
        // 转换坐标系
        m_p1.x = m_p1.x - 10;
        m_p1.y = 280 - m_p1.y;
        m_p2.x = m_p2.x - 10;
        m_p2.y = 280 - m_p2.y;

        // 计算斜率和截距
        m_fA = (float) (m_p2.y - m_p1.y) / (m_p2.x - m_p1.x);
        m_fB = m_p1.y - m_fA * m_p1.x;

        // 保存变动
        UpdateData(FALSE);
    }

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}
else
{
    // 用户在绘制区域外放开鼠标左键

    // 获取绘图的标签
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 获取设备上下文
    CDC* pDC = pWnd->GetDC();

    // 设置绘制方式为异或模式
    int nOldDrawMode = pDC->SetROP2(R2_XORPEN);

    // 创建新的画笔
    CPen* pPen = new CPen;
    pPen->CreatePen(PS_DOT, 1, RGB(0,0,0));

    // 选中新画笔
    CGdiObject* pOldPen = pDC->SelectObject(pPen);

    // 判断是否已经画过橡皮筋线
    if (m_bDrawed)
    {
        // 擦去以前的橡皮筋线
```

```
                pDC->MoveTo(m_p1);
                pDC->LineTo(m_p2);
            }

            // 选回以前的画笔
            pDC->SelectObject(pOldPen);

            // 恢复成以前的绘制模式
            pDC->SetROP2(nOldDrawMode);

            delete pPen;
            ReleaseDC(pDC);

        }

        // 解除对鼠标事件的跟踪
        ::ReleaseCapture();

        // 重置拖动状态
        m_bIsDraging = FALSE;
    }

    // 默认释放鼠标左键处理事件
    CDialog::OnLButtonUp(nFlags, point);
}

void CDlgLinerPara::OnPaint()
{
    // 字符串
    CString str;

    // 直线和坐标轴二个交点坐标
    int x1, y1, x2, y2;

    // 设备上下文
    CPaintDC dc(this);

    // 获取绘制坐标的文本框
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 指针
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();

    pDC->Rectangle(0,0,330,300);

    // 创建画笔对象
    CPen* pPenRed = new CPen;

    // 红色画笔
```

```
pPenRed->CreatePen(PS_SOLID,2,RGB(255,0,0));

// 创建画笔对象
CPen* pPenBlue = new CPen;

// 蓝色画笔
pPenBlue->CreatePen(PS_SOLID,2,RGB(0,0, 255));

// 选中当前红色画笔，并保存以前的画笔
CGdiObject* pOldPen = pDC->SelectObject(pPenRed);

// 绘制坐标轴
pDC->MoveTo(10,10);

// 垂直轴
pDC->LineTo(10,280);

// 水平轴
pDC->LineTo(320,280);

// 写坐标
str.Format("0");
pDC->TextOut(10, 281, str);

str.Format("255");
pDC->TextOut(265, 281, str);
pDC->TextOut(11, 25, str);

// 绘制X轴箭头
pDC->LineTo(315,275);
pDC->MoveTo(320,280);
pDC->LineTo(315,285);

// 绘制X轴箭头
pDC->MoveTo(10,10);
pDC->LineTo(5,15);
pDC->MoveTo(10,10);
pDC->LineTo(15,15);

// 更改成蓝色画笔
pDC->SelectObject(pPenBlue);

// 计算直线和坐标轴二个交点坐标
if (m_fA >= 0)
{
    if (((m_fA * 255 + m_fB) >= 0) && (m_fB < 255))
    {
        // 计算(x1, y1)坐标
        if (m_fB < 0)
        {
            x1 = (int) (- m_fB/m_fA + 0.5);
```

```
                y1 = 0;
            }
            else
            {
                x1 = 0;
                y1 = (int) (m_fB + 0.5);
            }

            // 计算(x2, y2)坐标
            if ((m_fA * 255 + m_fB) > 255)
            {
                x2 = (int) ((255- m_fB)/m_fA + 0.5);
                y2 = 255;
            }
            else
            {
                x2 = 255;
                y2 = (int) (255* m_fA + m_fB + 0.5);
            }
        }
        else if(((m_fA * 255 + m_fB) < 0))
        {
            // 转换后所有像素值都小于0，直接设置为0
            x1 = 0;
            y1 = 0;
            x2 = 255;
            y2 = 0;
        }
        else
        {
            // 转换后所有像素值都大于255，直接设置为255
            x1 = 0;
            y1 = 255;
            x2 = 255;
            y2 = 255;
        }
    }
    else  // 斜率小于0
    {
        if ((m_fB > 0) && (255* m_fA + m_fB < 255))
        {
            // 计算(x1, y1)坐标
            if (m_fB > 255)
            {
                x1 = (int) ((255- m_fB)/m_fA + 0.5);
                y1 = 255;
            }
            else
            {
                x1 = 0;
                y1 = (int) (m_fB + 0.5);
```

```
            }
            // 计算(x2, y2)坐标
            if ((m_fA * 255 + m_fB) < 0)
            {
                x2 = (int) (- m_fB/m_fA + 0.5);
                y2 = 0;
            }
            else
            {
                x2 = 255;
                y2 = (int) (255* m_fA + m_fB + 0.5);
            }
        }
        else if (m_fB <=0)
        {
            // 转换后所有像素值都小于0，直接设置为0
            x1 = 0;
            y1 = 0;
            x2 = 255;
            y2 = 0;
        }
        else
        {
            // 转换后所有像素值都大于255，直接设置为255
            x1 = 0;
            y1 = 255;
            x2 = 255;
            y2 = 255;
        }
    }
    // 绘制坐标值
    str.Format("(%d, %d)", x1, y1);
    pDC->TextOut(x1 + 10, 280 - y1 + 1, str);
    str.Format("(%d, %d)", x2, y2);
    pDC->TextOut(x2 + 10, 280 - y2 + 1, str);

    // 绘制用户指定的线性变换直线（注意转换坐标系）
    pDC->MoveTo(x1 + 10, 280 - y1);
    pDC->LineTo(x2 + 10, 280 - y2);

    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);

    // 绘制边缘
    pDC->MoveTo(10,25);
    pDC->LineTo(265,25);
    pDC->LineTo(265,280);
    // 删除新的画笔
    delete pPenRed;
    delete pPenBlue;
}
```

上述代码运行的结果如图 3−8 所示。

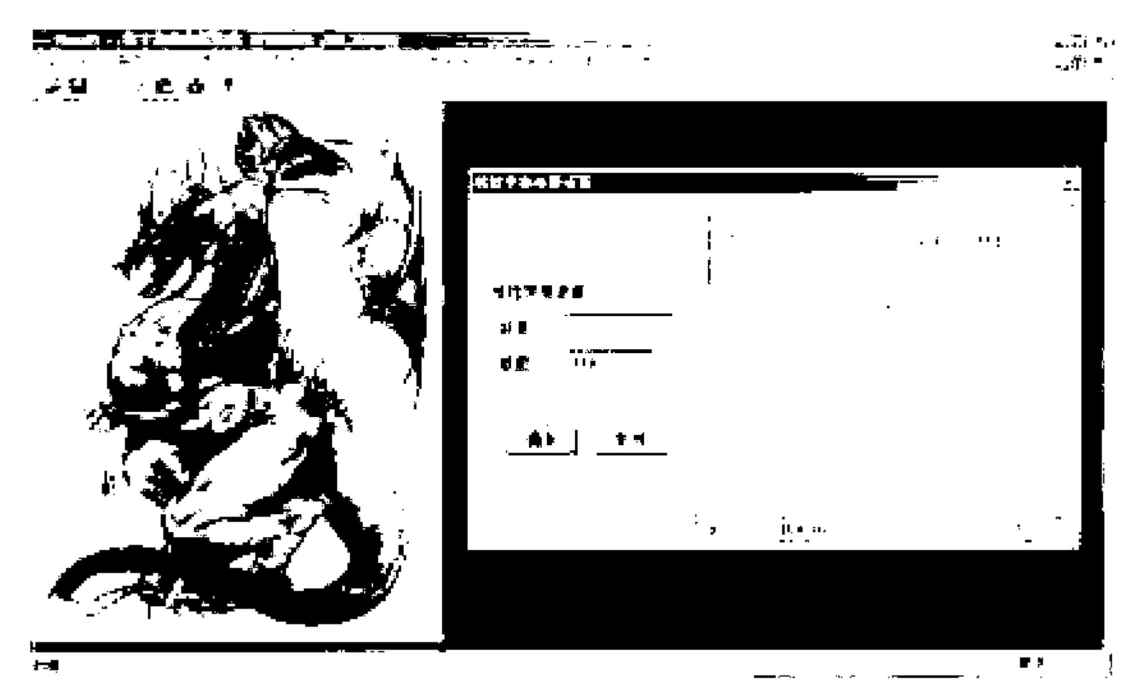


图 3−8 灰度线性变换

图像变换（$f(x)=2x-128$）后的灰度直方图如图 3−9 所示。原始图像的灰度直方图如图 3−10 所示。读者可以对比一下这两个图。

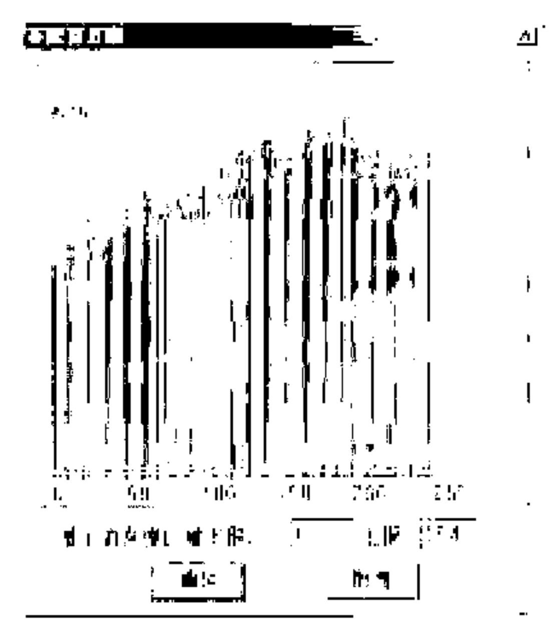


图 3−9 灰度线性变换后的直方图

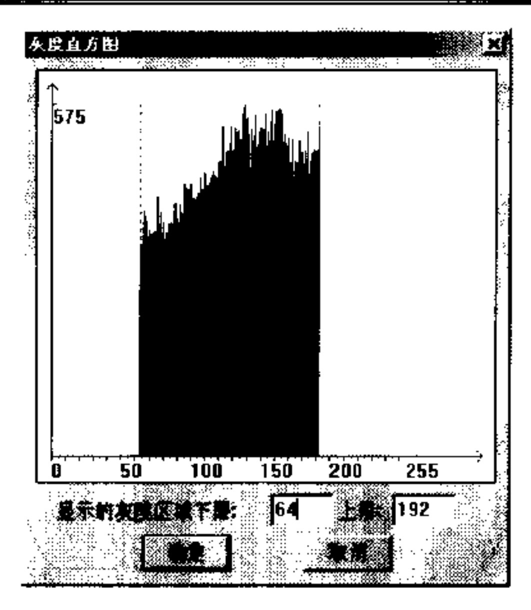


图 3－10　灰度线性变换前的直方图（显示区间为 64－192）

可见该变换其实就是把原始图像 64~192 灰度区间的点线性拉伸到 0~255 灰度区间，将灰度值小于 64 的像素灰度设置为 0，将灰度值大于 192 的像素灰度设置为 255，图像的对比度变化由此增强。

3.3　灰度的阈值变换

3.3.1　理论基础

灰度的阈值变换可以将一幅灰度图像转换成黑白二值图像。它的操作过程是先由用户指定一个阈值，如果图像中某像素的灰度值小于该阈值，则将该像素的灰度值设置为 0，否则灰度值设置为 255。

灰度阈值变换的变换函数表达式如下：

$$f(x)=\begin{cases}0 & x<T\\ 255 & x\geq T\end{cases}$$

其中 T 为指定的阈值。

3.3.2　Visual C++编程实现

下面代码中定义的 ThresholdTrans ()函数就是用来实现图像灰度的阈值变换。

```
/*************************************************************************
 *
 * 函数名称:
 *   ThresholdTrans()
 *
```

```
 * 参数:
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LONG  lWidth        - 原图像宽度（像素数）
 *   LONG  lHeight       - 原图像高度（像素数）
 *   BYTE  bThre         - 阈值
 *
 * 返回值:
 *   BOOL                - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行阈值变换。对于灰度值小于阈值的像素直接设置
 * 灰度值为0；灰度值大于阈值的像素直接设置为255。
 *
 ************************************************************************/
BOOL WINAPI ThresholdTrans(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BYTE bThre)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 判断是否小于阈值
            if ((*lpSrc) < bThre)
            {
                // 直接赋值为0
                *lpSrc = 0;
            }
            else
            {
                // 直接赋值为255
                *lpSrc = 255;
            }
        }
```

```
    }

    // 返回
    return TRUE;
}
```

在类 CCh1_1View 中添加图像阈值变换菜单项的事件处理程序：

```
void CCh1_1View::OnPointThre()
{
    // 阈值变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 参数对话框
    CDlgPointThre  dlgPara;

    // 阈值
    BYTE     bThre;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的阈值变换，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的阈值变换！", "系统提示" ,
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 初始化变量值
    dlgPara.m_bThre = 128;

    // 显示对话框，提示用户设定阈值
    if (dlgPara.DoModal() != IDOK)
```

```
	{
		// 返回
		return;
	}

	// 获取用户设定的阈值
	bThre = dlgPara.m_bThre;

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 调用ThresholdTrans()函数进行阈值变换
	::ThresholdTrans(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), bThre);

	// 设置脏标记
	pDoc->SetModifiedFlag(TRUE);

	// 更新视图
	pDoc->UpdateAllViews(NULL);

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();
}
```

上面代码中 CDlgThrePara 是一个新建的对话框类，该对话框主要是用来让用户设置阈值变换的参数：阈值 m_bThre。该对话框的完整代码如下：

1. 对话框头文件 DlgThrePara.h

```
#if !defined(AFX_DLGPOINTTHRE_H__4CF9C804_C248_4119_B6AD_905FB6AF4D89__INCLUDED_)
#define AFX_DLGPOINTTHRE_H__4CF9C804_C248_4119_B6AD_905FB6AF4D89__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgPointThre.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgPointThre dialog

class CDlgPointThre : public CDialog
{
// Construction
public:
```

```
    // 当前鼠标拖动状态，TRUE表示正在拖动
    BOOL    m_bIsDraging;

    // 相应鼠标事件的矩形区域
    CRect   m_MouseRect;

    CDlgPointThre(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(DlgPointThre)
    enum { IDD = IDD_DLG_PointThre };

    // 阈值
    BYTE    m_bThre;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(DlgPointThre)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(DlgPointThre)
    afx_msg void OnKillfocusEDITThre();
    virtual BOOL OnInitDialog();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnPaint();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_DLGPOINTTHRE_H__4CF9C804_C248_4119_B6AD_905FB6AF4D89__INCLUDED_)
```

2. 对话框代码 DlgThrePara.cpp

```
// DlgPointThre.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
```

```
#include "DlgPointThre.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// DlgPointThre dialog


CDlgPointThre::CDlgPointThre(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgPointThre::IDD, pParent)
{
    //{{AFX_DATA_INIT(DlgPointThre)
    m_bThre = 0;
    //}}AFX_DATA_INIT
}


void CDlgPointThre::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(DlgPointThre)
    DDX_Text(pDX, IDC_EDIT_Thre, m_bThre);
    DDV_MinMaxByte(pDX, m_bThre, 0, 255);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgPointThre, CDialog)
    //{{AFX_MSG_MAP(DlgPointThre)
    ON_EN_KILLFOCUS(IDC_EDIT_Thre, OnKillfocusEDITThre)
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    ON_WM_PAINT()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgPointThre message handlers

BOOL CDlgPointThre::OnInitDialog()
{

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();

    // 获取绘制直方图的标签
```

```
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 计算接受鼠标事件的有效区域
    pWnd->GetClientRect(m_MouseRect);
    pWnd->ClientToScreen(&m_MouseRect);

    CRect rect;
    GetClientRect(rect);
    ClientToScreen(&rect);

    m_MouseRect.top -= rect.top;
    m_MouseRect.left -= rect.left;

    // 设置接受鼠标事件的有效区域
    m_MouseRect.top += 25;
    m_MouseRect.left += 10;
    m_MouseRect.bottom = m_MouseRect.top + 255;
    m_MouseRect.right = m_MouseRect.left + 256;

    // 初始化拖动状态
    m_bIsDraging = FALSE;

    // 返回TRUE
    return TRUE;
}

void CDlgPointThre::OnKillfocusEDITThre()
{
    // 更新
    UpdateData(TRUE);

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointThre::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当用户单击鼠标左键开始拖动

    // 判断是否在有效区域中
    if(m_MouseRect.PtInRect(point))
    {
        if (point.x == (m_MouseRect.left + m_bThre))
        {

            // 设置拖动状态
            m_bIsDraging = TRUE;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
```

```
    }

    // 默认单击鼠标左键处理事件
    CDialog::OnLButtonDown(nFlags, point);
}

void CDlgPointThre::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 当用户释放鼠标左键停止拖动
    if (m_bIsDraging)
    {
        // 重置拖动状态
        m_bIsDraging = FALSE;
    }

    // 默认释放鼠标左键处理事件
    CDialog::OnLButtonUp(nFlags, point);
}

void CDlgPointThre::OnMouseMove(UINT nFlags, CPoint point)
{
    // 判断当前光标是否在绘制区域
    if(m_MouseRect.PtInRect(point))
    {
        // 判断是否正在拖动
        if (m_bIsDraging)
        {
            // 更改阈值
            m_bThre = (BYTE) (point.x - m_MouseRect.left);

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));

            // 更新
            UpdateData(FALSE);

            // 重绘
            InvalidateRect(m_MouseRect, TRUE);
        }
        else if (point.x == (m_MouseRect.left + m_bThre))
        {
            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
    }

    // 默认鼠标移动处理事件
    CDialog::OnMouseMove(nFlags, point);
}

void CDlgPointThre::OnPaint()
```

```
{
    // 字符串
    CString str;

    // 设备上下文
    CPaintDC dc(this);

    // 获取绘制坐标的文本框
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 指针
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();

    pDC->Rectangle(0,0,330,300);

    // 创建画笔对象
    CPen* pPenRed = new CPen;

    // 红色画笔
    pPenRed->CreatePen(PS_SOLID,2,RGB(255,0,0));

    // 创建画笔对象
    CPen* pPenBlue = new CPen;

    // 蓝色画笔
    pPenBlue->CreatePen(PS_SOLID,2,RGB(0,0, 255));

    // 创建画笔对象
    CPen* pPenGreen = new CPen;

    // 绿色画笔
    pPenGreen->CreatePen(PS_DOT,1,RGB(0,255,0));

    // 选中当前红色画笔，并保存以前的画笔
    CGdiObject* pOldPen = pDC->SelectObject(pPenRed);

    // 绘制坐标轴
    pDC->MoveTo(10,10);

    // 垂直轴
    pDC->LineTo(10,280);

    // 水平轴
    pDC->LineTo(320,280);

    // 写坐标
    str.Format("0");
    pDC->TextOut(10, 281, str);
```

```
    str.Format("255");
    pDC->TextOut(265, 281, str);
    pDC->TextOut(11, 25, str);

    // 绘制X轴箭头
    pDC->LineTo(315,275);
    pDC->MoveTo(320,280);
    pDC->LineTo(315,285);

    // 绘制X轴箭头
    pDC->MoveTo(10,10);
    pDC->LineTo(5,15);
    pDC->MoveTo(10,10);
    pDC->LineTo(15,15);

    // 更改成绿色画笔
    pDC->SelectObject(pPenGreen);

    // 绘制窗口阈值线
    pDC->MoveTo(m_bThre + 10, 25);
    pDC->LineTo(m_bThre + 10, 280);

    // 更改成蓝色画笔
    pDC->SelectObject(pPenBlue);

    // 绘制坐标值
    str.Format("%d", m_bThre);
    pDC->TextOut(m_bThre + 10, 281, str);

    // 绘制用户指定的窗口（注意转换坐标系）
    pDC->MoveTo(10, 280);
    pDC->LineTo(m_bThre + 10, 280);
    pDC->LineTo(m_bThre + 10, 25);
    pDC->LineTo(265, 25);

    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);

    // 绘制边缘
    pDC->MoveTo(10,25);
    pDC->LineTo(265,25);
    pDC->LineTo(265,280);

    // 删除新的画笔
    delete pPenRed;
    delete pPenBlue;
    delete pPenGreen;
}
```

上述代码运行的结果如图 3－11 所示。

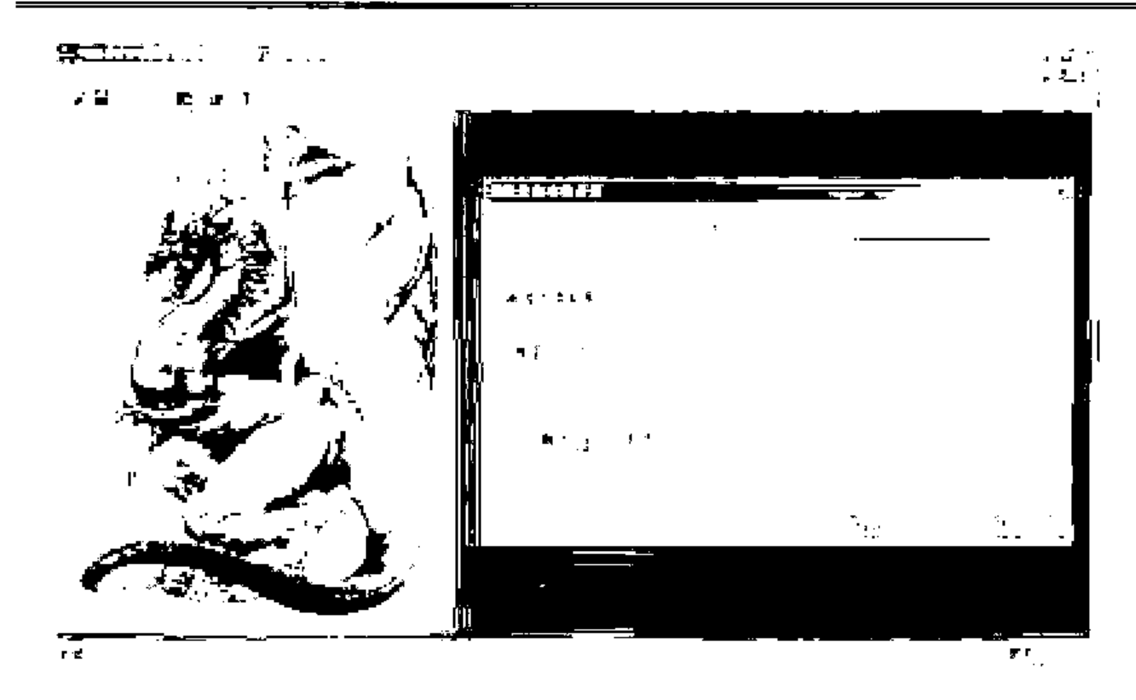


图 3－11　灰度阈值变换

3.4　灰度的窗口变换

3.4.1　理论基础

灰度的窗口变换也是一种常见的点运算。它的操作和阈值变换相类似。它限定一个窗口范围，该窗口中的灰度值保持不变；小于该窗口下限的灰度值直接设置为 0；大于该窗口上限的灰度值直接设置为 255。

灰度窗口变换的变换函数表达式如下：

$$f(x)=\begin{cases}0 & x<L\\ x & L\le x\le U\\ 255 & x>U\end{cases}$$

式中 L 表示窗口的下限，U 表示窗口的上限。

灰度窗口变换非常实用。例如：一幅图像的灰度直方图如图 3－12 所示，图像的背景是浅色，图像上的物体是深色，则直方图上的第一个峰值表示物体，第二个峰值表示背景。

设双峰之间的谷底在 T 处，当该图像进行窗口变换时，窗口上限取值为 T，下限为 0，变换后的结果将有效的消除图像的背景。

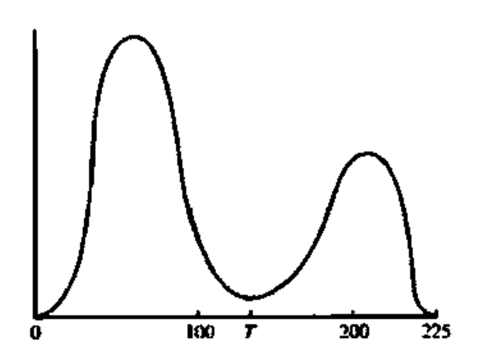


图 3－12 双峰直方图

图 3－13 所示图像的灰度直方图如图 3－14 所示。可以看出图像灰度为 0（黑色）处有一个峰值，灰度为 160 处是背景色的峰值。如果对该图像进行窗口变换，窗口下限取 0，上限取 100，则变换结果如图 3－15 所示。



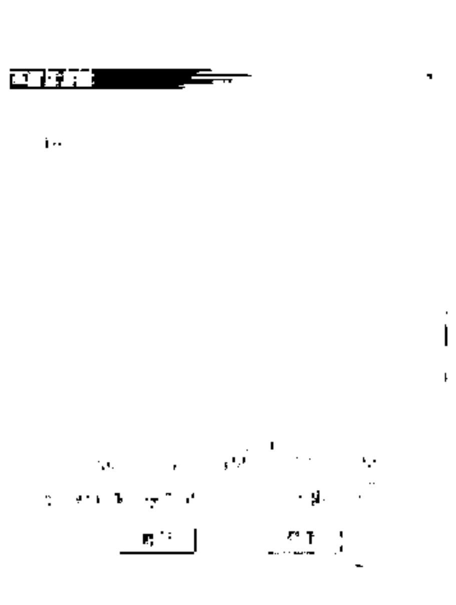


图 3－14 双峰直方图



图3－15 窗口变换后图像

可以看出图像的背景基本被消除了。

3.4.2 Visual C++编程实现

下面我们来编写灰度窗口变换的函数。和灰度的线性变换相似，窗口变换操作不需要改变 DIB 的调色板和文件头，只要把指向 DIB 像素起始位置的指针、DIB 高度宽度信息以及窗口上限和下限传递给子函数就可以完成灰度窗口变换工作。下面代码中定义的 WindowTrans() 函数就是用来实现图像灰度的窗口变换的。

```
/*************************************************************************
 *
 * 函数名称：
 *   WindowTrans()
 *
 * 参数：
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   BYTE  bLow         - 窗口下限
 *   BYTE  bUp          - 窗口上限
 *
 * 返回值：
 *   BOOL               - 成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用来对图像进行窗口变换。只有在窗口范围内的灰度保持不变，
 * 小于下限的像素直接设置灰度值为0；大于上限的像素直接设置灰度值为255。
 *
 ************************************************************************/
BOOL WINAPI WindowTrans(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BYTE bLow, BYTE bUp)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
```

```
    LONG    j;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 判断是否超出范围
            if ((*lpSrc) < bLow)
            {
                // 直接赋值为0
                *lpSrc = 0;
            }
            else if ((*lpSrc) > bUp)
            {
                // 直接赋值为255
                *lpSrc = 255;
            }
        }
    }

    // 返回
    return TRUE;
}
```

在类 CCh1_1View 中添加窗口变换菜单项的事件处理程序：

```
void CCh1_1View::OnPointWind()
{
    // 窗口变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 参数对话框
    CDlgPointWin  dlgPara;
```

```
// 窗口下限
BYTE    bLow;

// 窗口上限
BYTE    bUp;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的窗口变换，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的窗口变换！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 初始化变量值
dlgPara.m_bLow = 0;
dlgPara.m_bUp = 255;

// 显示对话框，提示用户设定窗口上下限
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户设定的窗口上下限
bLow = dlgPara.m_bLow;
bUp = dlgPara.m_bUp;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 调用WindowTrans()函数进行窗口变换
::WindowTrans(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), bLow, bUp);
```

```
    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
}
```

上面代码中 CDlgPointWin 是一个新建的对话框类，该对话框主要是用来让用户设置窗口变换的上下限：m_bLow 和 m_bUp。该对话框的完整代码如下：

1. 对话框头文件 DlgPointWin.h

```
#if !defined(AFX_DLGPOINTWIN_H__E76A76B4_7F31_421A_9EB3_2AD952C0F009__INCLUDED_)
#define AFX_DLGPOINTWIN_H__E76A76B4_7F31_421A_9EB3_2AD952C0F009__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgPointWin.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgPointWin dialog

class CDlgPointWin : public CDialog
{
// Construction
public:

    // 当前鼠标拖动状态，0表示未拖动，1表示正在拖动下限，2表示正在拖动上限。
    int     m_iIsDraging;

    // 相应鼠标事件的矩形区域
    CRect   m_MouseRect;

    CDlgPointWin(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgPointWin)
    enum { IDD = IDD_DLG_PointWin };

    // 窗口的下限
    BYTE    m_bLow;

    // 窗口的上限
    BYTE    m_bUp;
```

```
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgPointWin)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgPointWin)
    afx_msg void OnKillfocusEDITLow();
    afx_msg void OnKillfocusEDITUp();
    virtual void OnOK();
    afx_msg void OnPaint();
    virtual BOOL OnInitDialog();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_DLGPOINTWIN_H__E76A76B4_7F31_421A_9EB3_2AD952C0F009__INCLUDED_)
```

2. 对话框代码 DlgPointWin.cpp

```
// DlgPointWin.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgPointWin.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgPointWin dialog
```

```
CDlgPointWin::CDlgPointWin(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgPointWin::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgPointWin)
    m_bLow = 0;
    m_bUp = 0;
    //}}AFX_DATA_INIT
}


void CDlgPointWin::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgPointWin)
    DDX_Text(pDX, IDC_EDIT_Low, m_bLow);
    DDV_MinMaxByte(pDX, m_bLow, 0, 255);
    DDX_Text(pDX, IDC_EDIT_Up, m_bUp);
    DDV_MinMaxByte(pDX, m_bUp, 0, 255);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgPointWin, CDialog)
    //{{AFX_MSG_MAP(CDlgPointWin)
    ON_EN_KILLFOCUS(IDC_EDIT_Low, OnKillfocusEDITLow)
    ON_EN_KILLFOCUS(IDC_EDIT_Up, OnKillfocusEDITUp)
    ON_WM_PAINT()
    ON_WM_LBUTTONDOWN()
    ON_WM_LBUTTONUP()
    ON_WM_MOUSEMOVE()
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgPointWin message handlers

BOOL CDlgPointWin::OnInitDialog()
{

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();

    // 获取绘制直方图的标签
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 计算接受鼠标事件的有效区域
    pWnd->GetClientRect(m_MouseRect);
    pWnd->ClientToScreen(&m_MouseRect);

    CRect rect;
    GetClientRect(rect);
```

```
    ClientToScreen(&rect);

    m_MouseRect.top -= rect.top;
    m_MouseRect.left -= rect.left;

    // 设置接受鼠标事件的有效区域
    m_MouseRect.top += 25;
    m_MouseRect.left += 10;
    m_MouseRect.bottom = m_MouseRect.top + 255;
    m_MouseRect.right = m_MouseRect.left + 256;

    // 初始化拖动状态
    m_iIsDraging = 0;

    // 返回TRUE
    return TRUE;
}

void CDlgPointWin::OnKillfocusEDITLow()
{
    // 更新
    UpdateData(TRUE);

    // 判断是否下限超过上限
    if (m_bLow > m_bUp)
    {
        // 互换
        BYTE bTemp = m_bLow;
        m_bLow = m_bUp;
        m_bUp = bTemp;

        // 更新
        UpdateData(FALSE);
    }

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointWin::OnKillfocusEDITUp()
{
    // 更新
    UpdateData(TRUE);

    // 判断是否下限超过上限
    if (m_bLow > m_bUp)
    {
        // 互换
        BYTE bTemp = m_bLow;
        m_bLow = m_bUp;
        m_bUp = bTemp;
```

```
        // 更新
        UpdateData(FALSE);
    }

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointWin::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当用户单击鼠标左键开始拖动
    if(m_MouseRect.PtInRect(point))
    {
        if (point.x == (m_MouseRect.left + m_bLow))
        {

            // 设置拖动状态1，拖动下限
            m_iIsDraging = 1;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
        else if (point.x == (m_MouseRect.left + m_bUp))
        {

            // 设置拖动状态为2，拖动上限
            m_iIsDraging = 2;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
        }
    }

    // 默认单击鼠标左键处理事件
    CDialog::OnLButtonDown(nFlags, point);
}

void CDlgPointWin::OnMouseMove(UINT nFlags, CPoint point)
{
    // 判断当前光标是否在绘制区域
    if(m_MouseRect.PtInRect(point))
    {
        // 判断是否正在拖动
        if (m_iIsDraging != 0)
        {
            // 判断正在拖动上限还是下限
            if (m_iIsDraging == 1)
            {
                // 判断是否下限<上限
                if (point.x - m_MouseRect.left < m_bUp)
```

```
            {
                // 更改下限
                m_bLow = (BYTE) (point.x - m_MouseRect.left);
            }
            else
            {
                // 下限拖过上限，设置为上限-1
                m_bLow = m_bUp - 1;

                // 重设鼠标位置
                point.x = m_MouseRect.left + m_bUp - 1;
            }
        }
        else
        {
            // 正在拖动上限

            // 判断是否上限>下限
            if (point.x - m_MouseRect.left > m_bLow)
            {
                // 更改下限
                m_bUp = (BYTE) (point.x - m_MouseRect.left);
            }
            else
            {
                // 下限拖过上限，设置为下限+1
                m_bUp = m_bLow + 1;

                // 重设鼠标位置
                point.x = m_MouseRect.left + m_bLow + 1;
            }
        }

        // 更改光标
        ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));

        // 更新
        UpdateData(FALSE);

        // 重绘
        InvalidateRect(m_MouseRect, TRUE);
    }
    else if (point.x == (m_MouseRect.left + m_bLow)
            || point.x == (m_MouseRect.left + m_bUp))
    {
        // 更改光标
        ::SetCursor(::LoadCursor(NULL, IDC_SIZEWE));
    }
}

// 默认鼠标移动处理事件
```

```
    CDialog::OnMouseMove(nFlags, point);
}

void CDlgPointWin::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 当用户释放鼠标左键停止拖动
    if (m_iIsDraging != 0)
    {
        // 重置拖动状态
        m_iIsDraging = 0;
    }

    // 默认释放鼠标左键处理事件
    CDialog::OnLButtonUp(nFlags, point);
}

void CDlgPointWin::OnPaint()
{
    // 字符串
    CString str;

    // 设备上下文
    CPaintDC dc(this);

    // 获取绘制坐标的文本框
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 指针
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();

    pDC->Rectangle(0,0,330,300);

    // 创建画笔对象
    CPen* pPenRed = new CPen;

    // 红色画笔
    pPenRed->CreatePen(PS_SOLID,2,RGB(255,0,0));

    // 创建画笔对象
    CPen* pPenBlue = new CPen;

    // 蓝色画笔
    pPenBlue->CreatePen(PS_SOLID,2,RGB(0,0, 255));

    // 创建画笔对象
    CPen* pPenGreen = new CPen;

    // 绿色画笔
    pPenGreen->CreatePen(PS_DOT,1,RGB(0,255,0));
```

```
// 选中当前红色画笔，并保存以前的画笔
CGdiObject* pOldPen = pDC->SelectObject(pPenRed);

// 绘制坐标轴
pDC->MoveTo(10,10);

// 垂直轴
pDC->LineTo(10,280);

// 水平轴
pDC->LineTo(320,280);

// 写坐标
str.Format("0");
pDC->TextOut(10, 281, str);

str.Format("255");
pDC->TextOut(265, 281, str);
pDC->TextOut(11, 25, str);

// 绘制X轴箭头
pDC->LineTo(315,275);
pDC->MoveTo(320,280);
pDC->LineTo(315,285);

// 绘制X轴箭头
pDC->MoveTo(10,10);
pDC->LineTo(5,15);
pDC->MoveTo(10,10);
pDC->LineTo(15,15);

// 更改成绿色画笔
pDC->SelectObject(pPenGreen);

// 绘制窗口下限
pDC->MoveTo(m_bLow + 10, 25);
pDC->LineTo(m_bLow + 10, 280);

// 绘制窗口上限
pDC->MoveTo(m_bUp + 10, 25);
pDC->LineTo(m_bUp + 10, 280);

// 更改成蓝色画笔
pDC->SelectObject(pPenBlue);

// 绘制坐标值
str.Format("(%d, %d)", m_bLow, m_bLow);
pDC->TextOut(m_bLow + 10, 281 - m_bLow, str);
str.Format("(%d, %d)", m_bUp, m_bUp);
pDC->TextOut(m_bUp + 10, 281 - m_bUp, str);
```

```
    // 绘制用户指定的窗口（注意转换坐标系）
    pDC->MoveTo(10, 280);

    pDC->LineTo(m_bLow + 10, 280);
    pDC->LineTo(m_bLow + 10, 280 - m_bLow);
    pDC->LineTo(m_bUp + 10, 280 - m_bUp);
    pDC->LineTo(m_bUp + 10, 25);
    pDC->LineTo(265, 25);

    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);

    // 绘制边缘
    pDC->MoveTo(10,25);
    pDC->LineTo(265,25);
    pDC->LineTo(265,280);

    // 删除新创建的红色画笔
    delete pPenRed;

    // 删除新创建的蓝色画笔
    delete pPenBlue;

    // 删除新创建的绿色画笔
    delete pPenGreen;

}

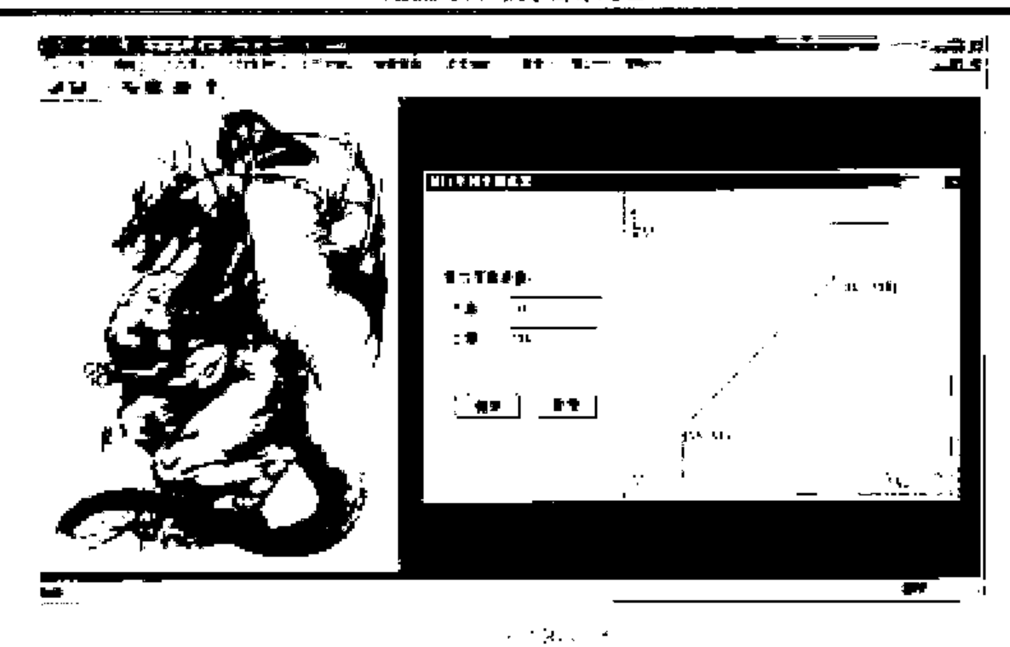
void CDlgPointWin::OnOK()
{

    // 判断是否下限超过上限
    if (m_bLow > m_bUp)
    {
        // 互换
        BYTE bTemp = m_bLow;
        m_bLow = m_bUp;
        m_bUp = bTemp;
    }

    // 默认处理事件
    CDialog::OnOK();
}
```

上述代码运行的结果如图 3－16 所示。

图像窗口变换（窗口下限为 50，上限为 200）后的灰度直方图如图 3－17 所示。读者可以把该图和原始图像的灰度直方图（图 3－4）对比一下，会发现两者不同之处在于变换后图像的灰度直方图 1－49 和 201~254 灰度范围内的像素数为 0。



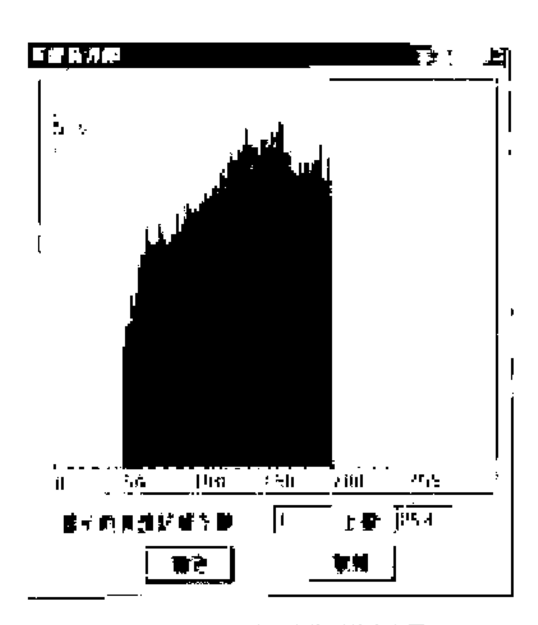


图 3－17 灰度窗口变换后的直方图

3.5 灰度拉伸

3.5.1 理论基础

灰度拉伸和灰度的线性变换有点类似，都用到了灰度的线性变换。但不同之处在于灰度拉伸不是完全的线性变换，而是分段进行线性变换。它的灰度变换函数如图 3－18 所示，函

数表达式如下：

$$f(x)=\begin{cases}\dfrac{y_1}{x_1}x & x<x_1\\ \dfrac{y_2-y_1}{x_2-x_1}(x-x_1)+y_1 & x_1\le x\le x_2\\ \dfrac{255-y_2}{255-x_2}(x-x_2)+y_2 & x>x_2\end{cases}$$

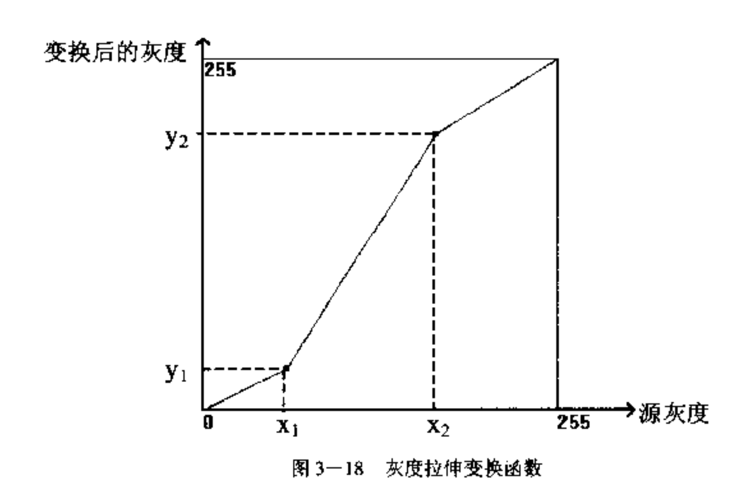


图 3－18 灰度拉伸变换函数

式中（x_1, y_1）和（x_2, y_2）是图 3－18 中的两个转折点的坐标。

灰度拉伸可以更加灵活的控制输出灰度直方图的分布，它可以有选择的拉伸某段灰度区间以改善输出图像。图 3－18 所示的变换函数的运算结果是将原图在 x_1 和 x_2 之间的灰度拉伸到 y_1 和 y_2 之间。如果一幅图像灰度集中在较暗的区域而导致图像偏暗，可以用灰度拉伸功能来拉伸（斜率>1）物体灰度区间以改善图像；同样如果图像灰度集中在较亮的区域而导致图像偏亮，也可以用灰度拉伸功能来压缩（斜率<1）物体灰度区间以改善图像质量。

3.5.2 Visual C++编程实现

下面我们来编写灰度拉伸变换的函数。和灰度的线性变换相似，灰度拉伸变换操作不需要改变 DIB 的调色板和文件头，只要把指向 DIB 像素起始位置的指针、DIB 高度宽度信息以及两个转折点坐标传递给子函数就可以完成灰度拉伸变换工作。下面代码定义的 GrayStretch() 函数就是用来实现图像灰度拉伸变换的。

```
/*************************************************************************
 *
 * 函数名称：
 *   GrayStretch()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
```

```
 *   BYTE bX1            - 灰度拉伸第一个点的X坐标
 *   BYTE bY1            - 灰度拉伸第一个点的Y坐标
 *   BYTE bX2            - 灰度拉伸第二个点的X坐标
 *   BYTE bY2            - 灰度拉伸第二个点的Y坐标
 *
 * 返回值:
 *   BOOL                - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行灰度拉伸。
 *
 ************************************************************************/
BOOL WINAPI GrayStretch(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BYTE bX1, BYTE bY1, BYTE
bX2, BYTE bY2)
{
    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 灰度映射表
    BYTE    bMap[256];

    // 图像每行的字节数
    LONG    lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 计算灰度映射表
    for (i = 0; i <= bX1; i++)
    {
        // 判断bX1是否大于0（防止分母为0）
        if (bX1 > 0)
        {
            // 线性变换
            bMap[i] = (BYTE) bY1 * i / bX1;
        }
        else
        {
            // 直接赋值为0
            bMap[i] = 0;
        }
    }
    for (; i <= bX2; i++)
    {
        // 判断bX1是否等于bX2（防止分母为0）
        if (bX2 != bX1)
        {
```

```
            // 线性变换
            bMap[i] = bY1 + (BYTE) ((bY2 - bY1) * (i - bX1) / (bX2 - bX1));
        }
        else
        {
            // 直接赋值为bY1
            bMap[i] = bY1;
        }
    }
    for (; i < 256; i++)
    {
        // 判断bX2是否等于255（防止分母为0）
        if (bX2 != 255)
        {
            // 线性变换
            bMap[i] = bY2 + (BYTE) ((255 - bY2) * (i - bX2) / (255 - bX2));
        }
        else
        {
            // 直接赋值为255
            bMap[i] = 255;
        }
    }

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 计算新的灰度值
            *lpSrc = bMap[*lpSrc];
        }
    }

    // 返回
    return TRUE;
}
```

在类 CCh1_1View 中添加拉伸变换菜单项的事件处理程序：

```
void CCh1_1View::OnPointStre()
{
    // 灰度拉伸

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
```

```
LPSTR    lpDIB;

// 指向DIB像素指针
LPSTR    lpDIBBits;

// 参数对话框
CDlgPointStre dlgPara;

// 点1坐标
BYTE     bX1;
BYTE     bY1;

// 点2坐标
BYTE     bX2;
BYTE     bY2;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的灰度拉伸，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的灰度拉伸！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 初始化变量值
dlgPara.m_bX1 = 50;
dlgPara.m_bY1 = 30;
dlgPara.m_bX2 = 200;
dlgPara.m_bY2 = 220;

// 显示对话框，提示用户设定拉伸位置
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户的设定
bX1 = dlgPara.m_bX1;
```

```
	bY1 = dlgPara.m_bY1;
	bX2 = dlgPara.m_bX2;
	bY2 = dlgPara.m_bY2;

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 调用GrayStretch()函数进行灰度拉伸
	::GrayStretch(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), bX1, bY1, bX2, bY2);

	// 设置脏标记
	pDoc->SetModifiedFlag(TRUE);

	// 更新视图
	pDoc->UpdateAllViews(NULL);

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();
}
```

上面代码中 CDlgPointStre 是一个新建的对话框类，该对话框主要是用来让用户设置拉伸变换的两个转折点坐标：m_bX1、m_bX 2、m_bY1 和 m_bY2。该对话框的完整代码如下：

1. 对话框头文件 DlgPointStre.h

```
#if !defined(AFX_DLGPOINTSTRE_H__45B95585_372F_4C49_8928_99D343D2DE00__INCLUDED_)
#define AFX_DLGPOINTSTRE_H__45B95585_372F_4C49_8928_99D343D2DE00__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgPointStre.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgPointStre dialog

class CDlgPointStre : public CDialog
{
// Construction
public:

	// 当前鼠标拖动状态，0表示未拖动，1表示正在拖动第一点，2表示正在拖动第二点。
	int		m_iIsDraging;

	// 相应鼠标事件的矩形区域
```

```
    CRect   m_MouseRect;

    // 标识是否已经绘制橡皮筋线
    BOOL    m_bDrawed;

    // 保存鼠标左键单击时的位置
    CPoint  m_p1;

    // 保存鼠标拖动时的位置
    CPoint  m_p2;

    CDlgPointStre(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgPointStre)
    enum { IDD = IDD_DLG_PointStre };

    // 两个转折点坐标
    BYTE    m_bX1;
    BYTE    m_bY1;
    BYTE    m_bX2;
    BYTE    m_bY2;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgPointStre)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgPointStre)
    virtual BOOL OnInitDialog();
    afx_msg void OnLButtonDown(UINT nFlags, CPoint point);
    afx_msg void OnMouseMove(UINT nFlags, CPoint point);
    afx_msg void OnLButtonUp(UINT nFlags, CPoint point);
    afx_msg void OnPaint();
    afx_msg void OnKillfocusEditX1();
    afx_msg void OnKillfocusEditX2();
    afx_msg void OnKillfocusEditY1();
    afx_msg void OnKillfocusEditY2();
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

```
//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_DLGPOINTSTRE_H__45B95585_372F_4C49_8928_99D343D2DE00__INCLUDED_)
```

2. 对话框代码 DlgPointStre.cpp

```
// DlgPointStre.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgPointStre.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgPointStre dialog


CDlgPointStre::CDlgPointStre(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgPointStre::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgPointStre)
    m_bX1 = 0;
    m_bY1 = 0;
    m_bX2 = 0;
    m_bY2 = 0;
    //}}AFX_DATA_INIT
}


void CDlgPointStre::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgPointStre)
    DDX_Text(pDX, IDC_EDIT_X1, m_bX1);
    DDV_MinMaxByte(pDX, m_bX1, 0, 255);
    DDX_Text(pDX, IDC_EDIT_X2, m_bX2);
    DDV_MinMaxByte(pDX, m_bY1, 0, 255);
    DDX_Text(pDX, IDC_EDIT_Y1, m_bY1);
    DDV_MinMaxByte(pDX, m_bX2, 0, 255);
    DDX_Text(pDX, IDC_EDIT_Y2, m_bY2);
    DDV_MinMaxByte(pDX, m_bY2, 0, 255);
    //}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CDlgPointStre, CDialog)
    //{{AFX_MSG_MAP(CDlgPointStre)
    ON_WM_LBUTTONDOWN()
    ON_WM_MOUSEMOVE()
    ON_WM_LBUTTONUP()
    ON_WM_PAINT()
    ON_EN_KILLFOCUS(IDC_EDIT_X1, OnKillfocusEditX1)
    ON_EN_KILLFOCUS(IDC_EDIT_X2, OnKillfocusEditX2)
    ON_EN_KILLFOCUS(IDC_EDIT_Y1, OnKillfocusEditY1)
    ON_EN_KILLFOCUS(IDC_EDIT_Y2, OnKillfocusEditY2)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgPointStre message handlers

BOOL CDlgPointStre::OnInitDialog()
{

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();

    // 获取绘制直方图的标签
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 计算接受鼠标事件的有效区域
    pWnd->GetClientRect(m_MouseRect);
    pWnd->ClientToScreen(&m_MouseRect);

    CRect rect;
    GetClientRect(rect);
    ClientToScreen(&rect);

    m_MouseRect.top -= rect.top;
    m_MouseRect.left -= rect.left;

    // 设置接受鼠标事件的有效区域
    m_MouseRect.top += 25;
    m_MouseRect.left += 10;
    m_MouseRect.bottom = m_MouseRect.top + 255;
    m_MouseRect.right = m_MouseRect.left + 256;

    // 初始化拖动状态
    m_iIsDraging = 0;

    // 返回TRUE
    return TRUE;
}

void CDlgPointStre::OnKillfocusEditX1()
```

```
{
    // 更新
    UpdateData(TRUE);

    // 判断是否下限超过上限
    if (m_bX1 > m_bX2)
    {
        // 互换
        BYTE bTemp = m_bX1;
        m_bX1 = m_bX2;
        m_bX2 = bTemp;
        bTemp = m_bY1;
        m_bY1 = m_bY2;
        m_bY2 = bTemp;

        // 更新
        UpdateData(FALSE);
    }

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointStre::OnKillfocusEditX2()
{
    // 更新
    UpdateData(TRUE);

    // 判断是否下限超过上限
    if (m_bX1 > m_bX2)
    {
        // 互换
        BYTE bTemp = m_bX1;
        m_bX1 = m_bX2;
        m_bX2 = bTemp;
        bTemp = m_bY1;
        m_bY1 = m_bY2;
        m_bY2 = bTemp;

        // 更新
        UpdateData(FALSE);
    }

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointStre::OnKillfocusEditY1()
{
    // 更新
    UpdateData(TRUE);
```

```
    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointStre::OnKillfocusEditY2()
{
    // 更新
    UpdateData(TRUE);

    // 重绘
    InvalidateRect(m_MouseRect, TRUE);
}

void CDlgPointStre::OnOK()
{
    // 判断是否下限超过上限
    if (m_bX1 > m_bX2)
    {
        // 互换
        BYTE bTemp = m_bX1;
        m_bX1 = m_bX2;
        m_bX2 = bTemp;
        bTemp = m_bY1;
        m_bY1 = m_bY2;
        m_bY2 = bTemp;

        // 更新
        UpdateData(FALSE);
    }

    // 默认处理事件
    CDialog::OnOK();
}

void CDlgPointStre::OnLButtonDown(UINT nFlags, CPoint point)
{
    // 当用户单击鼠标左键开始拖动
    if(m_MouseRect.PtInRect(point))
    {
        CRect   rectTemp;

        // 计算点1临近区域
        rectTemp.left = m_MouseRect.left + m_bX1 - 2;
        rectTemp.right = m_MouseRect.left + m_bX1 + 2;
        rectTemp.top = 255 + m_MouseRect.top - m_bY1 - 2;
        rectTemp.bottom = 255 + m_MouseRect.top - m_bY1 + 2;

        // 判断用户是不是拖动点1
        if (rectTemp.PtInRect(point))
        {
```

```
            // 设置拖动状态1，拖动点1
            m_iIsDraging = 1;

            // 更改光标
            ::SetCursor(::LoadCursor(NULL, IDC_SIZEALL));
        }
        else
        {
            // 计算点2临近区域
            rectTemp.left = m_MouseRect.left + m_bX2 - 2;
            rectTemp.right = m_MouseRect.left + m_bX2 + 2;
            rectTemp.top = 255 + m_MouseRect.top - m_bY2 - 2;
            rectTemp.bottom = 255 + m_MouseRect.top - m_bY2 + 2;

            // 判断用户是不是拖动点2
            if (rectTemp.PtInRect(point))
            {

                // 设置拖动状态为2，拖动点2
                m_iIsDraging = 2;

                // 更改光标
                ::SetCursor(::LoadCursor(NULL, IDC_SIZEALL));
            }
        }
    }

    // 默认单击鼠标左键处理事件
    CDialog::OnLButtonDown(nFlags, point);
}

void CDlgPointStre::OnMouseMove(UINT nFlags, CPoint point)
{
    // 判断当前光标是否在绘制区域
    if(m_MouseRect.PtInRect(point))
    {
        // 判断是否正在拖动
        if (m_iIsDraging != 0)
        {
            // 判断正在拖动点1还是点2
            if (m_iIsDraging == 1)
            {
                // 判断是否下限<上限
                if (point.x - m_MouseRect.left < m_bX2)
                {
                    // 更改下限
                    m_bX1 = (BYTE) (point.x - m_MouseRect.left);
                }
                else
                {
```

```
                // 下限拖过上限，设置为上限-1
                m_bX1 = m_bX2 - 1;

                // 重设鼠标位置
                point.x = m_MouseRect.left + m_bX2 - 1;
            }

            // 更改Y坐标
            m_bY1 = (BYTE) (255 + m_MouseRect.top - point.y);
        }
        else
        {
            // 正在拖动点2

            // 判断是否上限>下限
            if (point.x - m_MouseRect.left > m_bX1)
            {
                // 更改下限
                m_bX2 = (BYTE) (point.x - m_MouseRect.left);
            }
            else
            {
                // 下限拖过上限，设置为下限+1
                m_bX2 = m_bX1 + 1;

                // 重设鼠标位置
                point.x = m_MouseRect.left + m_bX1 + 1;
            }

            // 更改Y坐标
            m_bY2 = (BYTE) (255 + m_MouseRect.top - point.y);
        }
        // 更改光标
        ::SetCursor(::LoadCursor(NULL, IDC_SIZEALL));

        // 更新
        UpdateData(FALSE);

        // 重绘
        InvalidateRect(m_MouseRect, TRUE);
    }
    else
    {
        CRect   rectTemp1;
        CRect   rectTemp2;
        // 计算点1临近区域
        rectTemp1.left = m_MouseRect.left + m_bX1 - 2;
        rectTemp1.right = m_MouseRect.left + m_bX1 + 2;
        rectTemp1.top = 255 + m_MouseRect.top - m_bY1 - 2;
        rectTemp1.bottom = 255 + m_MouseRect.top - m_bY1 + 2;
```

```
            // 计算点2临近区域
            rectTemp2.left = m_MouseRect.left + m_bX2 - 2;
            rectTemp2.right = m_MouseRect.left + m_bX2 + 2;
            rectTemp2.top = 255 + m_MouseRect.top - m_bY2 - 2;
            rectTemp2.bottom = 255 + m_MouseRect.top - m_bY2 + 2;

            // 判断用户在点1或点2旁边
            if ((rectTemp1.PtInRect(point)) || (rectTemp2.PtInRect(point)))
            {
                // 更改光标
                ::SetCursor(::LoadCursor(NULL, IDC_SIZEALL));
            }
        }
    }
    // 默认鼠标移动处理事件
    CDialog::OnMouseMove(nFlags, point);
}

void CDlgPointStre::OnLButtonUp(UINT nFlags, CPoint point)
{
    // 当用户释放鼠标左键停止拖动
    if (m_iIsDraging != 0)
    {
        // 重置拖动状态
        m_iIsDraging = 0;
    }
    // 默认释放鼠标左键处理事件
    CDialog::OnLButtonUp(nFlags, point);
}

void CDlgPointStre::OnPaint()
{
    // 字符串
    CString str;
    // 设备上下文
    CPaintDC dc(this);
    // 获取绘制坐标的文本框
    CWnd* pWnd = GetDlgItem(IDC_COORD);

    // 指针
    CDC* pDC = pWnd->GetDC();
    pWnd->Invalidate();
    pWnd->UpdateWindow();
    pDC->Rectangle(0,0,330,300);

    // 创建画笔对象
    CPen* pPenRed = new CPen;

    // 红色画笔
    pPenRed->CreatePen(PS_SOLID, 2, RGB(255,0,0));
```

```
// 创建画笔对象
CPen* pPenBlue = new CPen;

// 蓝色画笔
pPenBlue->CreatePen(PS_SOLID, 1, RGB(0,0, 255));

// 选中当前红色画笔，并保存以前的画笔
CGdiObject* pOldPen = pDC->SelectObject(pPenRed);

// 绘制坐标轴
pDC->MoveTo(10,10);

// 垂直轴
pDC->LineTo(10,280);

// 水平轴
pDC->LineTo(320,280);

// 写坐标
str.Format("0");
pDC->TextOut(10, 281, str);

str.Format("255");
pDC->TextOut(265, 281, str);
pDC->TextOut(11, 25, str);

// 绘制X轴箭头
pDC->LineTo(315,275);
pDC->MoveTo(320,280);
pDC->LineTo(315,285);

// 绘制X轴箭头
pDC->MoveTo(10,10);
pDC->LineTo(5,15);
pDC->MoveTo(10,10);
pDC->LineTo(15,15);

// 更改成蓝色画笔
pDC->SelectObject(pPenBlue);
// 绘制坐标值
str.Format("(%d, %d)", m_bX1, m_bY1);
pDC->TextOut(m_bX1 + 10, 281 - m_bY1, str);
str.Format("(%d, %d)", m_bX2, m_bY2);
pDC->TextOut(m_bX2 + 10, 281 - m_bY2, str);

// 绘制用户指定的变换直线
pDC->MoveTo(10, 280);
pDC->LineTo(m_bX1 + 10, 280 - m_bY1);
pDC->LineTo(m_bX2 + 10, 280 - m_bY2);
pDC->LineTo(265, 25);
```

```
    // 绘制点边缘的小矩形
    CBrush  brush;
    brush.CreateSolidBrush(RGB(0,255,0));

    // 选中刷子
    CGdiObject* pOldBrush = pDC->SelectObject(&brush);
    // 绘制小矩形
    pDC->Rectangle(m_bX1 + 10 - 2, 280 - m_bY1 - 2, m_bX1 + 12, 280 - m_bY1 + 2);
    pDC->Rectangle(m_bX2 + 10 - 2, 280 - m_bY2 - 2, m_bX2 + 12, 280 - m_bY2 + 2);

    // 恢复以前的画笔
    pDC->SelectObject(pOldPen);

    // 绘制边缘
    pDC->MoveTo(10,25);
    pDC->LineTo(265,25);
    pDC->LineTo(265,280);

    // 删除新的画笔
    delete pPenRed;
    delete pPenBlue;
}
```

上述代码运行的结果如图 3－19 所示。



图 3－19 灰度窗口变换

图像 3－1 灰度拉伸（二个转折点坐标为 50, 100 和 200, 150）后的灰度直方图如图 3－20 所示。读者把该图和原始图像的灰度直方图（图 3－4）对比一下后可以发现原图像的 0~50 灰度区间被拉伸到 0~100；50~200 灰度区间被压缩到 100~150；200~255 灰度区间被拉伸到 150~255。

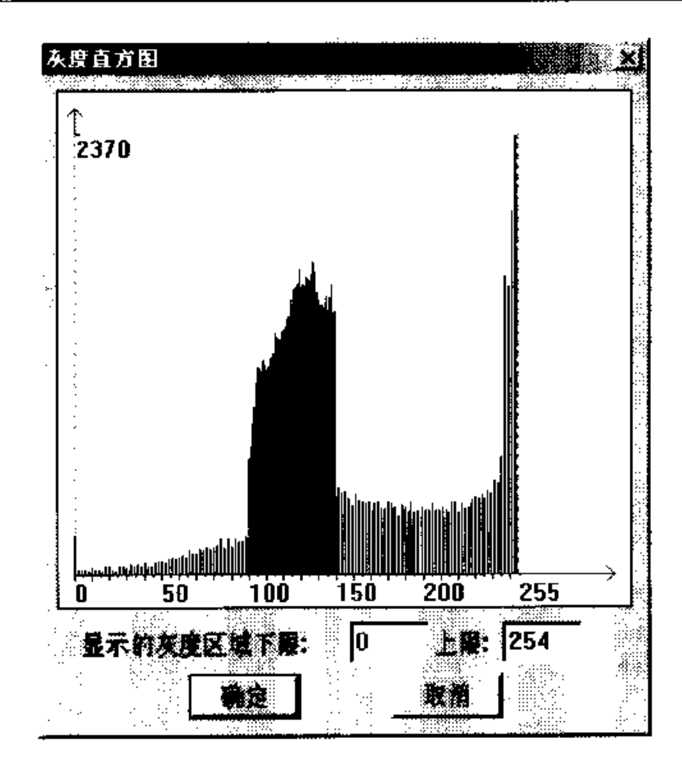


图 3-20 灰度窗口变换后的直方图

3.6 灰度均衡

3.6.1 理论基础

灰度均衡有时也称直方图均衡，目的是通过点运算使输入图像转换为在每一灰度级上都有相同的像素点数的输出图像（即输出的直方图是平的）。这对于在进行图像比较或分割之前将图像转化为一致的格式是十分有益的。

按照图像的概率密度函数（PDF，归一化到单位面积的直方图）的定义：

$$p(x)=\frac{1}{A_0}H(x)$$

其中 H(x)为直方图，A_0 为图像的面积。

设转换前图像的概率密度函数为 $p_r(r)$，转换后图像的概率密度函数为 $p_s(s)$，转换函数为 $s=f(r)$。由概率论知识，我们可以得到：

$$p_s(s)=p_r(r)\cdot\frac{\mathrm{d}r}{\mathrm{d}s}$$

这样，如果想使转换后图像的概率密度函数为 1（即直方图为平的），则必须满足：

$$p_r(r)=\frac{\mathrm{d}s}{\mathrm{d}r}$$

等式两边对 r 积分，可得：

$$s = f(r) = \int_0^r p_r(\mu)\mathrm{d}\mu = \frac{1}{A_0}\int_0^r H(\mu)\mathrm{d}\mu$$

该转换公式被称为图像的累积分布函数（CDF）。

上面的公式是被归一化后推导出的，对于没有归一化的情况，只要乘以最大灰度值(D_{Max}，对于灰度图就是 255）即可。灰度均衡的转换公式为：

$$D_B = f(D_A) = \frac{D_{\text{Max}}}{A_0}\int_0^{D_A} H(\mu)\mathrm{d}\mu$$

对于离散图像，转换公式为：

$$D_B = f(D_A) = \frac{D_{\text{Max}}}{A_0}\sum_{i=0}^{D_A} H_i$$

式中 H_i 为第 i 级灰度的像素个数。

3.6.2 Visual C++编程实现

根据上面的理论公式，现在我们来编写灰度均衡变换的函数。灰度均衡操作同样不需要改变 DIB 的调色板和文件头，只要把指向 DIB 像素起始位置的指针和 DIB 高度宽度信息传递给子函数就可以完成灰度均衡变换工作。下面代码定义的 InteEqualize()函数就是用来实现图像灰度均衡变换的。

```
/*************************************************************************
 *
 * 函数名称：
 *   InteEqualize()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行直方图均衡。
 *
 ************************************************************************/

BOOL WINAPI InteEqualize(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
```

```
// 指向原图像的指针
unsigned char*  lpSrc;

// 临时变量
LONG     lTemp;

// 循环变量
LONG     i;
LONG     j;

// 灰度映射表
BYTE     bMap[256];

// 灰度映射表
LONG     lCount[256];

// 图像每行的字节数
LONG     lLineBytes;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 重置计数为0
for (i = 0; i < 256; i ++)
{
    // 清零
    lCount[i] = 0;
}

// 计算各个灰度值的计数
for (i = 0; i < lHeight; i ++)
{
    for (j = 0; j < lWidth; j ++)
    {
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * i + j;

        // 计数加1
        lCount[*(lpSrc)]++;
    }
}

// 计算灰度映射表
for (i = 0; i < 256; i++)
{
    // 初始为0
    lTemp = 0;

    for (j = 0; j <= i ; j++)
    {
        lTemp += lCount[j];
    }
```

```
        // 计算对应的新灰度值
        bMap[i] = (BYTE) (lTemp * 255 / lHeight / lWidth);
    }

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 计算新的灰度值
            *lpSrc = bMap[*lpSrc];
        }
    }

    // 返回
    return TRUE;
}
```

在类 CCh1_1View 中添加灰度均衡菜单项的事件处理程序：

```
void CCh1_1View::OnPointEqua()
{

    // 灰度均衡

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的直方图均衡，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {

        // 提示用户
        MessageBox("目前只支持256色位图的直方图均衡！", "系统提示",
            MB_ICONINFORMATION | MB_OK);
```

```
        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 调用InteEqualize()函数进行直方图均衡
    ::InteEqualize(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB));

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

图 3－21 是原始图像及其灰度直方图，图 3－22 是利用上述代码进行灰度均衡后的图像和它的灰度直方图。

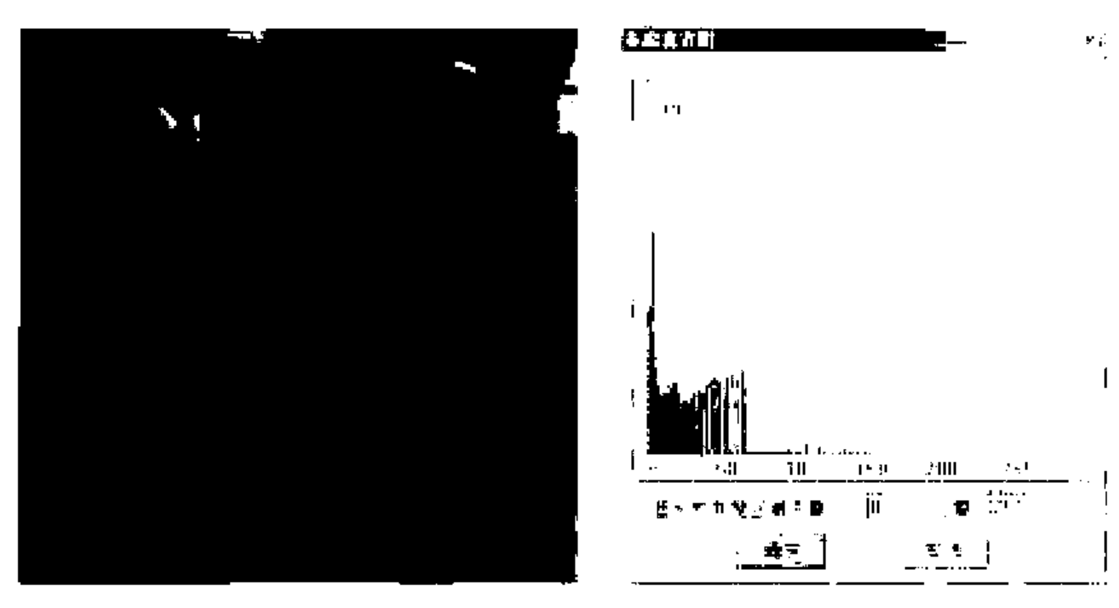


图 3－21 原始图像及其灰度直方图



图 3－22 灰度均衡后图像及其灰度直方图

可见，经过灰度均衡后，图像的对比度大大提高，转换后图像的灰度分布也趋于均匀。由于灰度级的个数有限，转换后图像实际的直方图不是理论上的一条水平直线，而是呈现出参差不齐的外形。

第四章　图像的几何变换

在上一章中我们介绍了图像的点运算。在本章中将介绍图像的另外一种基本变换：几何变换。它通常包括图像的平移、图像的镜象变换、图像的转置、图像的缩放和图像的旋转等。在介绍每种几何变换时，将先介绍该种变换的理论基础（主要是一些矩阵运算），接下来才介绍如何在 Visual C++中编程实现该变换。

4.1　图像的平移

图像的平移是几何变换中最简单的变换之一。下面介绍一下有关图像平移的理论基础。

4.1.1　理论基础

图像平移就是将图像中所有的点都按照指定的平移量水平、垂直移动。如图 4－1 所示，设$(x0, y0)$为原图像上的一点，图像水平平移量为 tx，垂直平移量为 ty，则平移后点$(x0, y0)$坐标将变为$(x1, y1)$。

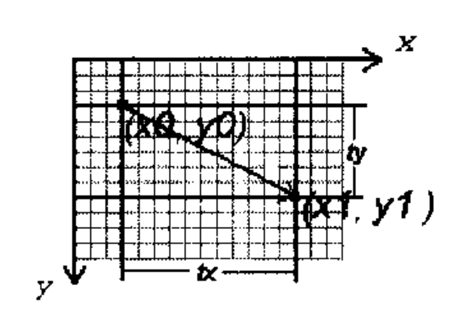


图 4－1　图像平移示意图

显然$(x0, y0)$和$(x1, y1)$的关系如下：

$$\begin{cases} x1 = x0 + tx \\ y1 = y0 + ty \end{cases}$$

用矩阵表示如下：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

对该矩阵求逆，可以得到逆变换：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -tx \\ 0 & 1 & -ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} \quad 即 \quad \begin{cases} x0 = x1 - tx \\ y0 = y1 - ty \end{cases}$$

这样，平移后的图像上的每一点都可以在原图像中找到对应的点。例如，对于新图中的(0,0)像素，代入上面的方程组，可以求出对应原图中的像素(-tx,-ty)。如果 tx 或 ty 大于 0，则点(-tx,-ty)不在原图中。对于不在原图中的点，可以直接将它的像素值统一设置为 0 或者 255（对于灰度图就是黑色或白色）。同样，若有点不在原图中，也就说明原图中有点被移出显示区域。如果不想丢失被移出的部分图像，可以将新生成的图像宽度扩大|tx|，高度扩大|ty|。

图 4－2 是没平移过的图像；图 4－3 是水平垂直都平移 100 像素后的图像；图 4－4 是平移扩大原图后的图像。





图 4－3 向右下平移后的图像（图像大小未变）



图 4-4 向右下平移后的图像（相应扩大图像）

4.1.2 Visual C++编程实现

有了上面的理论基础，我们可以十分容易地用 Visual C++来实现图像的平移。在这里，我们只介绍灰度图像的平移，是因为灰度图像每个像素位数正好是 8 位，即 1 个字节，这样，在进行图像处理时可以不用考虑拼凑字节的问题。而且由于灰度图调色板的特殊性，进行灰度图像处理时不必考虑调色板的问题。这样在介绍图像处理时，一般采用灰度图，为的是将重点放在算法本身。今后给出的程序如不做特殊说明，都是针对 256 级灰度图的，其他颜色的情况，可以很容易地类推出来。

由上面的公式我们可以得到如下算法：

```
// 每行
for(i = 0; i < lHeight; i++)
{
    // 每列
    for(j = 0; j < lWidth; j++)
    {
        // 指向新DIB第i行，第j个像素的指针

        // 注意由于DIB中图像第一行其实保存在最后一行的位置，因此lpDst
        // 值不是(char *)lpNewDIBBits + lLineBytes* i + j，而是
        // (char *)lpNewDIBBits + lLineBytes* (lHeight - 1 - i) + j
        lpDst = (char *)lpNewDIBBits + lLineBytes* (lHeight - 1 - i) + j;

        // 计算该像素在原DIB中的坐标
        i0 = i - lXOffset;
        j0 = j - lYOffset;

        // 判断是否在原图范围内
```

```
            if( (j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))
            {
                // 指向原DIB第i0行，第j0个像素的指针

                // 同样要注意DIB上下倒置的问题
                lpSrc = (char *)lpDIBBits + lLineBytes* (lHeight - 1 - i0) + j0;

                // 复制像素
                *lpDst = *lpSrc;
            }
            else
            {
                // 对于原图中没有的像素，直接赋值为255
                * ((unsigned char*)lpDst) = 255;
            }

        }
    }

    // 复制平移后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lLineBytes* lHeight);
```

由于每行像素是连续放置的，我们也可以直接逐行地来复制图像。首先计算出移动后可视的区域。

对于 x 轴方向，

- 当 tx≤-width 时，图像完全移出了屏幕，不用做任何处理；
- 当-width<tx≤0 时，图像区域的 x 范围从 0 到 width-|tx|，对应原图的范围从|tx|到 width；
- 当 0<tx<width 时，图像区域的 x 范围从 tx 到 width，对应原图的范围从 0 到 width-tx；
- 当 tx≥width 时，图像完全移出了屏幕，不用做任何处理；

对于 y 轴方向，

- 当 ty≤-height 时，图像完全移出了屏幕，不用做任何处理；
- 当-height<ty≤0 时，图像区域的 y 范围从 0 到 height-|ty|,对应原图的范围从|ty|到 height；
- 当 0<ty<height 时，图像区域的 y 范围从 ty 到 height,对应原图的范围从 0 到 height-ty；
- 当 ty≥height 时，图像完全移出了屏幕，不用做任何处理；

当计算出经移动而可视的区域后，就可以利用位图存储的连续性，即同一行的像素在内存中是相邻的这一规则进行计算。利用 memcpy 函数，从($x0$,$y0$)点开始，一次可以拷贝一整行（宽度为 $x1$-$x0$），然后将内存指针移到($x0$,$y0$+1)处，拷贝下一行。这样拷贝到($y1$-$y0$)行就完成了全部操作，避免了单个像素的计算，从而提高了效率。

按照上面的描述，我们现在可以构造自己的图像几何变换函数库了。首先我们来完成图像的平移函数。图像的平移函数操作不需要改变 DIB 的调色板和文件头，只要把指向 DIB 像素起始位置的指针和 DIB 高宽传递给子函数就可以完成平移工作。下面代码中定义的 TranslationDIB1()函数和 TranslationDIB()函数就是分别利用上面介绍的两种方法来平移 DIB 位图的。

```
/*************************************************************************
 *  文件名：GeoTrans.cpp
 *
 *  图像几何变换API函数库：
 *
 *  TranslationDIB1()   - 图像平移
 *  TranslationDIB()    - 图像平移
 *  MirrorDIB()         - 图像镜像
 *  TransposeDIB()      - 图像转置
 *  ZoomDIB()           - 图像缩放
 *  RotateDIB()         - 图像旋转
 *
 *************************************************************************/

#include "stdafx.h"
#include "geotrans.h"
#include "DIBAPI.h"

#include <math.h>
#include <direct.h>

/*************************************************************************
 *
 * 函数名称：
 *   TranslationDIB1()
 *
 * 参数：
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   LONG  lXOffset     - X轴平移量（像素数）
 *   LONG  lYOffset     - Y轴平移量（像素数）
 *
 * 返回值：
 *   BOOL               - 平移成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用来水平移动DIB图像。函数不会改变图像的大小，移出的部分图像
 * 将截去，空白部分用白色填充。
 *
 ************************************************************************/

BOOL WINAPI TranslationDIB1(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, LONG lXOffset, LONG
lYOffset)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向要复制区域的指针
    LPSTR   lpDst;
```

```
// 指向复制图像的指针
LPSTR   lpNewDIBBits;
HLOCAL  hNewDIBBits;

// 像素在新DIB中的坐标
LONG    i;
LONG    j;

// 像素在原DIB中的坐标
LONG    i0;
LONG    j0;

// 图像每行的字节数
LONG lLineBytes;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lLineBytes * lHeight);
if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 每行
for(i = 0; i < lHeight; i++)
{
    // 每列
    for(j = 0; j < lWidth; j++)
    {
        // 指向新DIB第i行，第j个像素的指针
        // 注意由于DIB中图像第一行其实保存在最后一行的位置，因此lpDst
        // 值不是(char *)lpNewDIBBits + lLineBytes * i + j，而是
        // (char *)lpNewDIBBits + lLineBytes * (lHeight - 1 - i) + j
        lpDst = (char *)lpNewDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        // 计算该像素在原DIB中的坐标
        i0 = i - lXOffset;
        j0 = j - lYOffset;

        // 判断是否在原图范围内
        if( (j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))
        {
            // 指向原DIB第i0行，第j0个像素的指针
            // 同样要注意DIB上下倒置的问题
            lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - 1 - i0) + j0;
```

```
                // 复制像素
                *lpDst = *lpSrc;
            }
            else
            {
                // 对于原图中没有的像素，直接赋值为255
                * ((unsigned char*)lpDst) = 255;
            }

        }
    }

    // 复制平移后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lLineBytes * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   TranslationDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   LONG  lXOffset     - X轴平移量（像素数）
 *   LONG  lYOffset     - Y轴平移量（像素数）
 *
 * 返回值:
 *   BOOL               - 平移成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来水平移动DIB图像。函数不会改变图像的大小，移出的部分图像
 * 将截去，空白部分用白色填充。
 *
 ************************************************************************/

BOOL WINAPI TranslationDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, LONG lXOffset, LONG
lYOffset)
{
    // 平移后剩余图像在原图像中的位置（矩形区域）
    CRect   rectSrc;
```

```
// 平移后剩余图像在新图像中的位置（矩形区域）
CRect    rectDst;

// 指向原图像的指针
LPSTR    lpSrc;

// 指向要复制区域的指针
LPSTR    lpDst;

// 指向复制图像的指针
LPSTR    lpNewDIBBits;
HLOCAL   hNewDIBBits;

// 指明图像是否全部移去可视区间
BOOL     bVisible;

// 循环变量
LONG     i;

// 图像每行的字节数
LONG lLineBytes;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 赋初值
bVisible = TRUE;

// 计算rectSrc和rectDst的X坐标
if (lXOffset <= -lWidth)
{
    // X轴方向全部移出可视区域
    bVisible = FALSE;
}
else if (lXOffset <= 0)
{
    // 移动后，有图区域左上角X坐标为0
    rectDst.left = 0;

    // 移动后，有图区域右下角X坐标为lWidth - |lXOffset| = lWidth + lXOffset
    rectDst.right = lWidth + lXOffset;
}
else if (lXOffset < lWidth)
{
    // 移动后，有图区域左上角X坐标为lXOffset
    rectDst.left = lXOffset;

    // 移动后，有图区域右下角X坐标为lWidth
    rectDst.right = lWidth;
}
else
```

```
{
    // X轴方向全部移出可视区域
    bVisible = FALSE;
}

// 平移后剩余图像在原图像中的X坐标
rectSrc.left = rectDst.left - lXOffset;
rectSrc.right = rectDst.right - lXOffset;

//  计算rectSrc和rectDst的Y坐标
if (lYOffset <= -lHeight)
{
    // Y轴方向全部移出可视区域
    bVisible = FALSE;
}
else if (lYOffset <= 0)
{
    // 移动后，有图区域左上角Y坐标为0
    rectDst.top = 0;

    // 移动后，有图区域右下角Y坐标为lHeight - |lYOffset| = lHeight + lYOffset
    rectDst.bottom = lHeight + lYOffset;
}
else if (lYOffset < lHeight)
{
    // 移动后，有图区域左上角Y坐标为lYOffset
    rectDst.top = lYOffset;

    // 移动后，有图区域右下角Y坐标为lHeight
    rectDst.bottom = lHeight;
}
else
{
    // X轴方向全部移出可视区域
    bVisible = FALSE;
}

// 平移后剩余图像在原图像中的Y坐标
rectSrc.top = rectDst.top - lYOffset;
rectSrc.bottom = rectDst.bottom - lYOffset;

// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lLineBytes * lHeight);

// 判断是否内存分配失败
if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
```

```
    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lLineBytes * lHeight);

    // 如果有部分图像可见
    if (bVisible)
    {
        // 平移图像
        for(i = 0; i < (rectSrc.bottom - rectSrc.top); i++)
        {
            // 要复制区域的起点，注意由于DIB图像内容是上下倒置的，第一行内容保存在最后
            // 一行，因此复制区域的起点不是lpDIBBits + lLineBytes * (i + rectSrc.top) +
            // rectSrc.left，而是 lpDIBBits + lLineBytes * (lHeight - i - rectSrc.top - 1)+
            // rectSrc.left。

            lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - i - rectSrc.top - 1) +
                    rectSrc.left;

            // 要目标区域的起点
            // 同样注意上下倒置的问题。
            lpDst = (char *)lpNewDIBBits + lLineBytes * (lHeight - i - rectDst.top - 1) +
                rectDst.left;

            // 拷贝每一行，宽度为rectSrc.right - rectSrc.left
            memcpy(lpDst, lpSrc, rectSrc.right - rectSrc.left);

        }
    }

    // 复制平移后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lLineBytes * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

对应的头文件内容如下：

```
// geotrans.h

#ifndef _INC_GeoTransAPI
#define _INC_GeoTransAPI

// 常数π
#define PI 3.1415926535
```

```
//角度到弧度转化的宏
#define RADIAN(angle) ((angle)*PI/180.0)

// 函数原型
BOOL WINAPI TranslationDIB1 (LPSTR lpDIBBits, LONG dwWidth, LONG dwHeight, LONG dwXOffset,
LONG dwYOffset);
BOOL WINAPI TranslationDIB (LPSTR lpDIBBits, LONG dwWidth, LONG dwHeight, LONG dwXOffset,
LONG dwYOffset);
BOOL WINAPI MirrorDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bDirection);
BOOL WINAPI TransposeDIB(LPSTR lpbi);
HGLOBAL WINAPI ZoomDIB(LPSTR lpbi, float fXZoomRatio, float fYZoomRatio);
HGLOBAL WINAPI RotateDIB(LPSTR lpbi, int iRotateAngle);
HGLOBAL WINAPI RotateDIB2(LPSTR lpbi, int iRotateAngle);
unsigned char WINAPI Interpolation (LPSTR lpDIBBits, LONG lWidth, LONG lHeight, FLOAT x, FLOAT
y);

#endif //!_INC_GeoTransAPI
```

完成图像平移函数后，我们可以更改第二章中完成的读写 DIB 图像的程序，以添加图像平移功能。首先添加一个名为“几何变换”的菜单项，如图 4-5 所示。

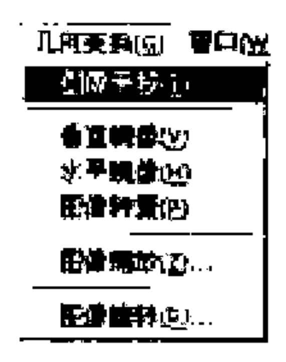


图 4-5　几何变换菜单项

在该菜单的事件处理函数中，我们添加如下代码：

```
void CCh1_1View::OnGeomTran()
{
    // 平移位图

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());
```

```
// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的平移，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{

    // 提示用户
    MessageBox("目前只支持256色位图的平移！", "系统提示", MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

LONG lXOffset;
LONG lYOffset;

// 创建对话框
CDlgGeoTran dlgPara;

// 初始化变量值
dlgPara.m_XOffset = 100;
dlgPara.m_YOffset = 100;

// 显示对话框，提示用户设定平移量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户设定的平移量
lXOffset = dlgPara.m_XOffset;
lYOffset = dlgPara.m_YOffset;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 调用TranslationDIB()或TranslationDIB1()函数平移DIB
if (::TranslationDIB1(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB),
      lXOffset, lYOffset))
{

    // 设置脏标记
```

```
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();

    }
```

上述代码中创建了一个 CDlgGeoTran dlgPara 对话框，该对话框是自己添加到资源中的，它主要用来让用户设置平移量。该对话框的代码比较简单，这里就不再列出其源代码。运行上面的代码进行平移的结果如图 4－6 所示。

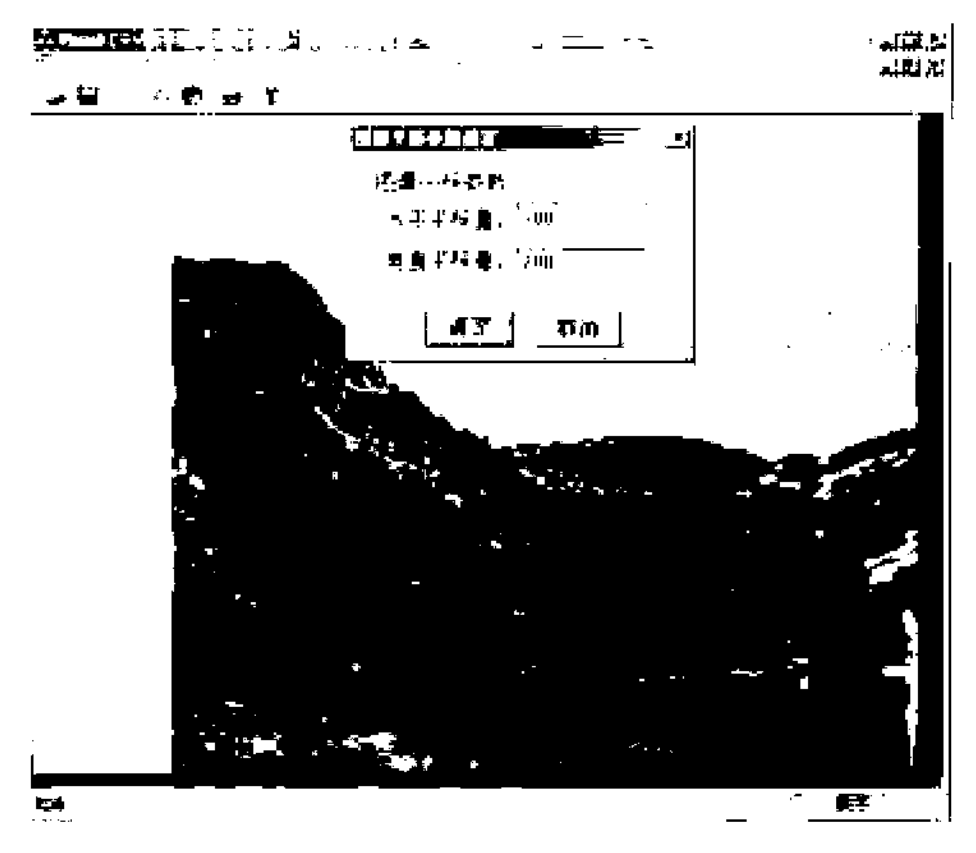


图 4－6　图像的平移

4.2 图像的镜像变换

图像的镜像（Mirror）变换分为两种：一种是水平镜像，另外一种是垂直镜像。图像的水平镜像操作是将图像左半部分和右半部分以图像垂直中轴线为中心镜像进行对换；图像的垂直镜像操作是将图像上半部分和下半部分以图像水平中轴线为中心镜像进行对换。下面我们看看数学中是如何描述该操作的。

4.2.1 理论基础

设图像高度为 *lHeight*，宽度为 *lWidth*，原图中$(x0,y0)$经过水平镜像后坐标将变为$(lWidth - x0,y0)$，其矩阵表达式为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & lWidth \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

逆运算矩阵表达式为：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} -1 & 0 & lWidth \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} \quad 即 \quad \begin{cases} x0 = lWidth - x1 \\ y0 = y1 \end{cases}$$

同样，$(x0,y0)$经过垂直镜像后坐标将变为$(x0, lHeight - y0)$，其矩阵表达式为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & lHeight \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

逆运算矩阵表达式为：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & lHeight \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} \quad 即 \quad \begin{cases} x0 = x1 \\ y0 = lHeight - y1 \end{cases}$$

4.2.2 Visual C++编程实现

按照上面的变换公式，我们可以非常简单的实现图像的水平和垂直镜像操作。代码如下所示（bDirection 为真时表示水平镜像，否则为垂直镜像）：

```
// 每行
for(i = 0; i < lHeight; i++)
{
    // 每列
    for(j = 0; j < lWidth; j++)
    {
        // 指向新DIB第i行，第j个像素的指针
```

```
        // 注意由于DIB中图像第一行其实保存在最后一行的位置，因此lpDst
        // 值不是(char *)lpNewDIBBits + lLineBytes* i + j，而是
        // (char *)lpNewDIBBits + lLineBytes* (lHeight - 1 - i) + j

        lpDst = (char *)lpNewDIBBits + lLineBytes* (lHeight - 1 - i) + j;

        // 计算该像素在原DIB中的坐标
        if (bDirection)
        {
            // 水平镜像
            i0 = i;
            j0 = lWidth - j;
        }
        else
        {
            // 垂直镜像
            i0 = lHeight - i;
            j0 = j;
        }

        // 水平和垂直镜像不会超出原图范围，因此不用判断
        // if( (j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))
        // {
            // 指向原DIB第i0行，第j0个像素的指针
            // 同样要注意DIB上下倒置的问题

            lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - 1 - i0) + j0;

            // 复制像素
            *lpDst = *lpSrc;

        // }
        // else
        // {
        //   // 对于原图中没有的像素，直接赋值为255
        //   * ((unsigned char*)lpDst) = 255;
        // }

    }
}

// 复制平移后的图像
memcpy(lpDIBBits, lpNewDIBBits, lLineBytes * lHeight);
```

和图像平移一样，在垂直镜像中也可以利用位图存储的连续性整行复制图像。在最终的图像几何变换函数库中，我们将采用这种的算法。下面代码就是图像镜像操作函数（函数MirrorDIB）的源代码。

```
/*************************************************************************
 *
 * 函数名称:
 *   MirrorDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   BOOL  bDirection   - 镜像的方向，TRUE表示水平镜像，FALSE表示垂直镜像
 *
 * 返回值:
 *   BOOL               - 镜像成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来镜像DIB图像。可以指定镜像的方式是水平还是垂直。
 *
 ************************************************************************/

BOOL WINAPI MirrorDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bDirection)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向要复制区域的指针
    LPSTR   lpDst;

    // 指向复制图像的指针
    LPSTR   lpBits;
    HLOCAL  hBits;

    // 循环变量
    LONG    i;
    LONG    j;

    // 图像每行的字节数
    LONG lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 暂时分配内存，以保存一行图像
    hBits = LocalAlloc(LHND, lLineBytes);

    // 判断是否内存分配失败
    if (hBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }
```

```
// 锁定内存
lpBits = (char * )LocalLock(hBits);

// 判断镜像方式
if (bDirection)
{
    // 水平镜像

    // 针对图像每行进行操作
    for(i = 0; i < lHeight; i++)
    {

        // 针对每行图像左半部分进行操作
        for(j = 0; j < lWidth / 2; j++)
        {

            // 指向倒数第i行，第j个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * i + j;

            // 指向倒数第i行，倒数第j个像素的指针
            lpDst = (char *)lpDIBBits + lLineBytes * (i + 1) - j;

            // 备份一个像素
            *lpBits = *lpDst;

            // 将倒数第i行，第j个像素复制到倒数第i行，倒数第j个像素
            *lpDst = *lpSrc;

            // 将倒数第i行，倒数第j个像素复制到倒数第i行，第j个像素
            *lpSrc = *lpBits;

        }

    }
}
else
{
    // 垂直镜像

    // 针对上半图像进行操作
    for(i = 0; i < lHeight / 2; i++)
    {

        // 指向倒数第i行像素起点的指针
        lpSrc = (char *)lpDIBBits + lLineBytes * i;

        // 指向第i行像素起点的指针
        lpDst = (char *)lpDIBBits + lLineBytes * (lHeight - i - 1);

        // 备份一行，宽度为lWidth
```

```
            memcpy(lpBits, lpDst, lLineBytes);

            // 将倒数第i行像素复制到第i行
            memcpy(lpDst, lpSrc, lLineBytes);

            // 将第i行像素复制到倒数第i行
            memcpy(lpSrc, lpBits, lLineBytes);

        }
    }

    // 释放内存
    LocalUnlock(hBits);

    LocalFree(hBits);

    // 返回
    return TRUE;

}
```

同样在 CCh1_1View 中添加相应菜单事件处理代码：

```
void CCh1_1View::OnGeomMirv()
{
    // 垂直镜像


    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的垂直镜像，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的垂直镜像！", "系统提示" ,
                MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
```

```
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用MirrorDIB()函数垂直镜像DIB
    if (::MirrorDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), FALSE))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}

void CCh1_1View::OnGeomMirh()
{
    // 水平镜像

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的水平镜像，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
```

```
    {

        // 提示用户
        MessageBox("目前只支持256色位图的水平镜像！", "系统提示",
                MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;

    }


    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用MirrorDIB()函数水平镜像DIB
    if (::MirrorDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), TRUE))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

图4—2中图像水平镜像结果如图4—7所示，垂直镜像结果如图4—8所示。





图 4—8 图像的垂直镜像

4.3 图像的转置

图像的转置（Transpose）操作是将图像像素的 x 坐标和 y 坐标互换。该操作将改变图像的大小：图像的高度和宽度将互换。下面我们看看数学中是如何描述该操作的。

4.3.1 理论基础

转置的变换矩阵表达式很简单：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

它的逆变换矩阵表达式是：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} \quad 即 \quad \begin{cases} x0 = y1 \\ y0 = x1 \end{cases}$$

4.3.2 Visual C++编程实现

图像转置的实现和图像镜像变换类似，不同之处在于图像转置后 DIB 的头文件也要进行相应的改变，主要是将头文件中图像高度和宽度信息更新。因此传递给图像转置函数的参数是直接指向 DIB 的指针，而不是直接指向 DIB 像素的指针。下面给出图像转置函数 TransposeDIB()的源代码。

```
/*************************************************************************
 *
 * 函数名称：
 *   TransposeDIB()
 *
 * 参数：
 *   LPSTR lpDIB          - 指向源DIB的指针
 *
 * 返回值：
 *   BOOL                 - 转置成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用来转置DIB图像，即图像x、y坐标互换。函数将不会改变图像的大小，
 * 但是图像的高宽将互换。
 *
 ************************************************************************/

BOOL WINAPI TransposeDIB(LPSTR lpDIB)
{

    // 图像的宽度和高度
    LONG    lWidth;
    LONG    lHeight;

    // 指向源图像的指针
    LPSTR   lpDIBBits;

    // 指向源像素的指针
    LPSTR   lpSrc;
```

```
// 指向转置图像对应像素的指针
LPSTR    lpDst;

// 指向转置图像的指针
LPSTR    lpNewDIBBits;
HLOCAL   hNewDIBBits;

// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFOHEADER lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREHEADER lpbmc;

// 循环变量
LONG     i;
LONG     j;

// 图像每行的字节数
LONG lLineBytes;

// 新图像每行的字节数
LONG lNewLineBytes;

// 获取指针
lpbmi = (LPBITMAPINFOHEADER)lpDIB;
lpbmc = (LPBITMAPCOREHEADER)lpDIB;

// 找到原DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 获取图像的"宽度"（4的倍数）
lWidth = ::DIBWidth(lpDIB);

// 获取图像的高度
lHeight = ::DIBHeight(lpDIB);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 计算新图像每行的字节数
lNewLineBytes = WIDTHBYTES(lHeight * 8);

// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lWidth * lNewLineBytes);

// 判断是否内存分配失败
if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
```

```
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 针对图像每行进行操作
    for(i = 0; i < lHeight; i++)
    {
        // 针对每行图像每列进行操作
        for(j = 0; j < lWidth; j++)
        {

            // 指向原DIB第i行，第j个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 指向转置DIB第j行，第i个像素的指针
            // 注意此处lWidth和lHeight是原DIB的宽度和高度，应该互换
            lpDst = (char *)lpNewDIBBits + lNewLineBytes * (lWidth - 1 - j) + i;

            // 复制像素
            *lpDst = *lpSrc;

        }

    }

    // 复制转置后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lNewLineBytes);

    // 互换DIB中图像的高宽
    if (IS_WIN30_DIB(lpDIB))
    {
        // 对于Windows 3.0 DIB
        lpbmi->biWidth = lHeight;

        lpbmi->biHeight = lWidth;
    }
    else
    {
        // 对于其他格式的DIB
        lpbmc->bcWidth = (unsigned short) lHeight;
        lpbmc->bcHeight = (unsigned short) lWidth;
    }

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

同样，在 CCh1_1View 中添加相应菜单事件处理代码：

```
void CCh1_1View::OnGeomTrpo()
{
    // 图像转置

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的转置，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的转置！", "系统提示", MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }


    // 更改光标形状
    BeginWaitCursor();

    // 调用TransposeDIB()函数转置DIB
    if (::TransposeDIB(lpDIB))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新DIB大小和调色板
        pDoc->InitDIBData();

        // 重新设置滚动视图大小
        SetScrollSizes(MM_TEXT, pDoc->GetDocSize());

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
```

```
        MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

图 4－2 中图像转置的结果如图 4－9 所示。



图 4－9　图像的转置

✍ **图像转置和下面将介绍的图像旋转是不同的，仅通过图像旋转是不可能实现图像转置的。旋转操作必须结合镜像操作才能实现图像的转置：首先水平镜像，然后逆时针旋转 90 度才可以实现。**

4.4　图像的缩放

上面介绍的几种图像几何变换中都是 1:1 的变换，本节和下节中介绍的图像变换将涉及到图像缩放和旋转操作。这些操作产生的图像中的像素可能在原图中找不到相应的像素点，

这样就必须进行近似处理。一般的方法是直接赋值为和它最相近的像素值，也可以通过一些插值算法来计算。后者处理效果要好些，但是运算量也相应增加很多。在下面的代码中我们直接采用了前一种做法（也是一种插值算法，称为最邻近插值，Nearest Neighbour Interpolation）。

4.4.1 理论基础

假设图像 X 轴方向缩放比率 fx，Y 轴方向缩放比率是 fy，那么原图中点 $(x0, y0)$ 对应与新图中的点 $(x1, y1)$ 的转换矩阵为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} fx & 0 & 0 \\ 0 & fy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

其逆运算如下：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1/fx & 0 & 0 \\ 0 & 1/fy & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} \quad 即 \quad \begin{cases} x0 = x1/fx \\ y0 = y1/fy \end{cases}$$

例如，当 $fx=fy=0.5$ 时，图像被缩到一半大小，此时缩小后图像中的(0, 0)像素对应于原图中的(0, 0)像素；(0, 1)像素对应于原图中的(0, 2)像素；(1, 0)像素对应于原图中的(2, 0)像素，以此类推。在原图基础上，每行隔一个像素取一点，每隔一行进行操作。同理，当 $fx=fy=2$ 时，图像放大 2 倍，放大后图像中的(0, 0)像素对应于原图中的(0, 0)像素；(0, 1)像素对应于原图中的(0, 0.5)像素，该像素不存在，可以近似为(0, 0)也可以近似为(0, 1)；(0, 2)像素对应于原图中的(0, 1)像素；(1, 0)像素对应于原图中的(0.5, 0) ≈ (0, 0)或(1, 0)像素；(2, 0)像素对应于原图中的(1, 0)像素，以此类推。其实是将原图每行中的像素重复取值一遍，然后每行重复一次。图 4－10 是原始图像，图 4－11 是分别采用两种近似方法放大后的图像。

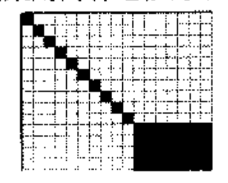
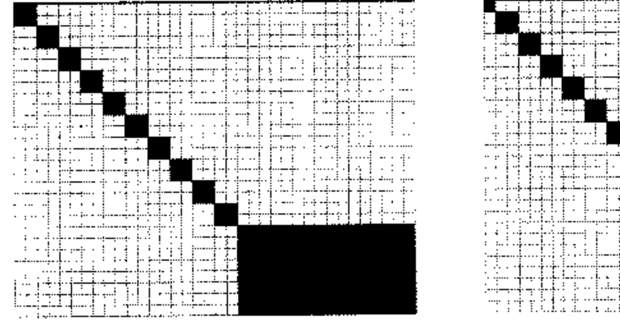


图 4－10 未放大的图像



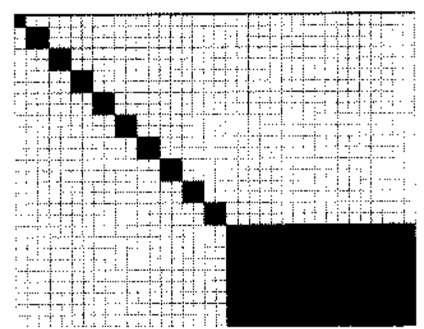


图 4－11 放大 2 倍后的图像

4.4.2 Visual C++编程实现

按照上面的转换公式，我们可以编写图像缩放函数 ZoomDIB ()，下面给出它的源代码。

```
/*************************************************************************
 *
 * 函数名称:
 *   ZoomDIB()
 *
 * 参数:
 *   LPSTR lpDIB        - 指向原DIB的指针
 *   float fXZoomRatio  - X轴方向缩放比率
 *   float fYZoomRatio  - Y轴方向缩放比率
 *
 * 返回值:
 *   HGLOBAL            - 缩放成功返回新DIB句柄，否则返回NULL。
 *
 * 说明:
 *   该函数用来缩放DIB图像，返回新生成DIB的句柄。
 *
 ************************************************************************/

HGLOBAL WINAPI ZoomDIB(LPSTR lpDIB, float fXZoomRatio, float fYZoomRatio)
{

    // 原图像的宽度和高度
    LONG    lWidth;
    LONG    lHeight;

    // 缩放后图像的宽度和高度
    LONG    lNewWidth;
    LONG    lNewHeight;

    // 缩放后图像的宽度（lNewWidth'，必须是4的倍数）
    LONG    lNewLineBytes;

    // 指向原图像的指针
    LPSTR   lpDIBBits;

    // 指向原像素的指针
    LPSTR   lpSrc;

    // 缩放后新DIB句柄
    HDIB    hDIB;

    // 指向缩放图像对应像素的指针
    LPSTR   lpDst;

    // 指向缩放图像的指针
    LPSTR   lpNewDIB;
    LPSTR   lpNewDIBBits;
```

```
// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFOHEADER lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREHEADER lpbmc;

// 循环变量（像素在新DIB中的坐标）
LONG    i;
LONG    j;

// 像素在原DIB中的坐标
LONG    i0;
LONG    j0;

// 图像每行的字节数
LONG lLineBytes;

// 找到原DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 获取图像的宽度
lWidth = ::DIBWidth(lpDIB);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 获取图像的高度
lHeight = ::DIBHeight(lpDIB);

// 计算缩放后的图像实际宽度
// 此处直接加0.5是由于强制类型转换时不四舍五入，而是直接截去小数部分
lNewWidth = (LONG) (::DIBWidth(lpDIB) * fXZoomRatio + 0.5);
lNewLineBytes = WIDTHBYTES(lNewWidth * 8);

// 计算缩放后的图像高度
lNewHeight = (LONG) (lHeight * fYZoomRatio + 0.5);

// 分配内存，以保存新DIB
hDIB = (HDIB) ::GlobalAlloc(GHND, lNewLineBytes * lNewHeight + *(LPDWORD)lpDIB
        + ::PaletteSize(lpDIB));

// 判断是否内存分配失败
if (hDIB == NULL)
{
    // 分配内存失败
    return NULL;
}

// 锁定内存
lpNewDIB =  (char * )::GlobalLock((HGLOBAL) hDIB);
```

```
// 复制DIB信息头和调色板
memcpy(lpNewDIB, lpDIB, *(LPDWORD)lpDIB + ::PaletteSize(lpDIB));

// 找到新DIB像素起始位置
lpNewDIBBits = ::FindDIBBits(lpNewDIB);

// 获取指针
lpbmi = (LPBITMAPINFOHEADER)lpNewDIB;
lpbmc = (LPBITMAPCOREHEADER)lpNewDIB;

// 更新DIB中图像的高度和宽度
if (IS_WIN30_DIB(lpNewDIB))
{
    // 对于Windows 3.0 DIB
    lpbmi->biWidth = lNewWidth;
    lpbmi->biHeight = lNewHeight;
}
else
{
    // 对于其他格式的DIB
    lpbmc->bcWidth = (unsigned short) lNewWidth;
    lpbmc->bcHeight = (unsigned short) lNewHeight;
}

// 针对图像每行进行操作
for(i = 0; i < lNewHeight; i++)
{
    // 针对图像每列进行操作
    for(j = 0; j < lNewWidth; j++)
    {

        // 指向新DIB第i行，第j个像素的指针
        // 注意此处宽度和高度是新DIB的宽度和高度
        lpDst = (char *)lpNewDIBBits + lNewLineBytes * (lNewHeight - 1 - i) + j;

        // 计算该像素在原DIB中的坐标
        i0 = (LONG) (i / fYZoomRatio + 0.5);
        j0 = (LONG) (j / fXZoomRatio + 0.5);

        // 判断是否在原图范围内
        if( (j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))
        {

            // 指向原DIB第i0行，第j0个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - 1 - i0) + j0;

            // 复制像素
            *lpDst = *lpSrc;
        }
        else
        {
```

```
                // 对于原图中没有的像素，直接赋值为255
                * ((unsigned char*)lpDst) = 255;
            }

        }

    }

    // 返回
    return hDIB;
}
```

同样在 CCh1_1View 中添加相应菜单事件处理代码：

```
void CCh1_1View::OnGeomZoom()
{
    // 图像缩放

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8 bpp位图（这里为了方便，只处理8-bpp位图的缩放，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {

        // 提示用户
        MessageBox("目前只支持256色位图的缩放！", "系统提示", MB_ICONINFORMATION  MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 缩放比率
    float fXZoomRatio;
    float fYZoomRatio;

    // 创建对话框
    CDlgGeoZoom dlgPara;

    // 初始化变量值
    dlgPara.m_XZoom = 0.5;
    dlgPara.m YZoom = 0.5;
```

```
// 显示对话框，提示用户设定缩放量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}


// 获取用户设定的平移量

// 水平缩放量
fXZoomRatio = dlgPara.m_XZoom;

// 垂直缩放量
fYZoomRatio = dlgPara.m_YZoom;

// 删除对话框
delete dlgPara;

// 创建新DIB
HDIB hNewDIB = NULL;

// 更改光标形状
BeginWaitCursor();

// 调用ZoomDIB()函数转置DIB
hNewDIB = (HDIB) ::ZoomDIB(lpDIB, fXZoomRatio, fYZoomRatio);

// 判断缩放是否成功
if (hNewDIB != NULL)
{

    // 替换DIB，同时释放旧DIB对象
    pDoc->ReplaceHDIB(hNewDIB);

    // 更新DIB大小和调色板
    pDoc->InitDIBData();

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 重新设置滚动视图大小
    SetScrollSizes(MM_TEXT, pDoc->GetDocSize());

    // 更新视图
    pDoc->UpdateAllViews(NULL);
}
else
{
    // 提示用户
    MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
```

```
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

上述代码中创建了一个 CDlgGeoZoom dlgPara 对话框，该对话框是自己添加到资源中的，它主要用来让用户设置水平和垂直缩放量。图 4－2 缩小一半的结果如图 4－12 所示。

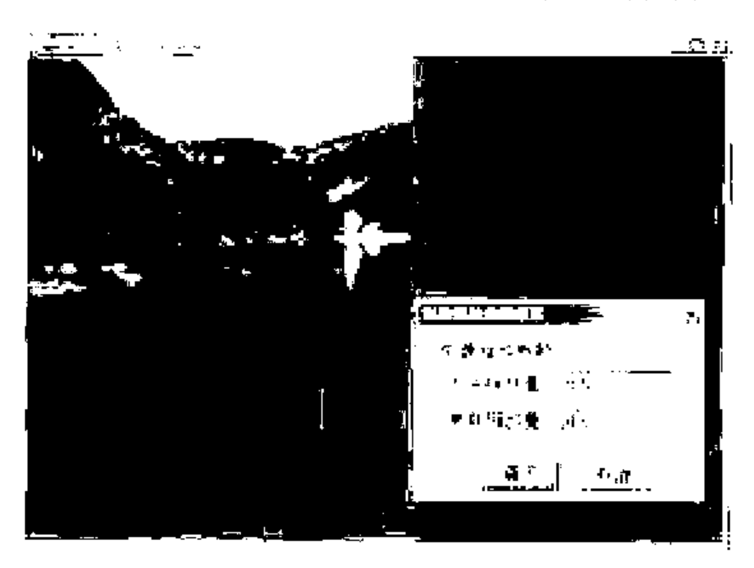


图 4－12　图像的缩放

4.5　图像的旋转

在本节中，我们将要介绍一种相对复杂的几何变换：图像的旋转。一般图像的旋转是以图像的中心为原点，旋转一定的角度。旋转后，图像的大小一般会改变。和图像平移一样，我们既可以把转出显示区域的图像截去，也可以扩大图像范围以显示所有的图像。如图 4－13、图 4－14、图 4－15 所示。

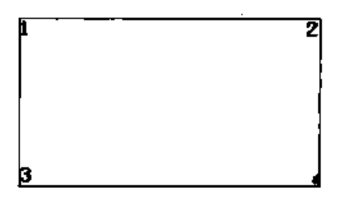


图 4－13　旋转前的图像

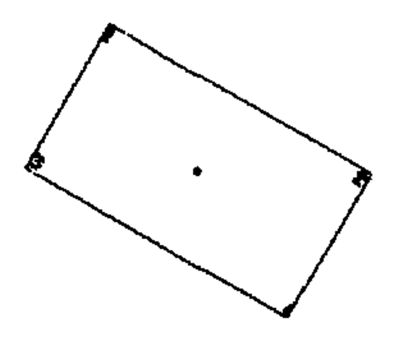


图4－14 旋转θ后的图像（扩大图像）

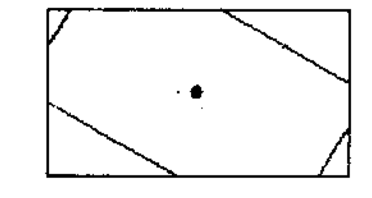


图4－15 旋转θ后的图像（转出部分被截去）

4.5.1 理论基础

下面我们来推导一下旋转运算的变换公式。如图4－16所示，点$(x0, y0)$经过旋转θ度后坐标变成$(x1, y1)$。

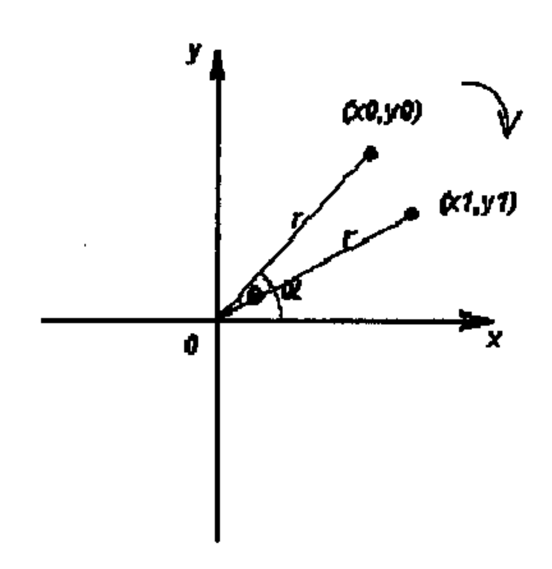


图4－16 旋转前的图像

在旋转前：

$$\begin{cases} x0 = r\cos(\alpha) \\ y0 = r\sin(\alpha) \end{cases}$$

旋转后：

$$\begin{cases} x1 = r\cos(\alpha-\theta) = r\cos(\alpha)\cos(\theta) + r\sin(\alpha)\sin(\theta) = x0\cos(\theta) + y0\sin(\theta) \\ y1 = r\sin(\alpha-\theta) = r\sin(\alpha)\cos(\theta) - r\cos(\alpha)\sin(\theta) = -x0\sin(\theta) + y0\cos(\theta) \end{cases}$$

写成矩阵表达式为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

其逆运算如下：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

上述旋转是绕坐标轴原点（0, 0）进行的，如果是绕一个指定点（a, b）旋转，则先要将坐标系平移到该点，再进行旋转，然后平移回新的坐标原点。

下面我们首先推导坐标系平移的转换公式。如图 4－17 所示，将坐标系 I 平移到坐标系 II 处，其中坐标系 II 的原点在坐标系 I 中坐标为（a, b）。

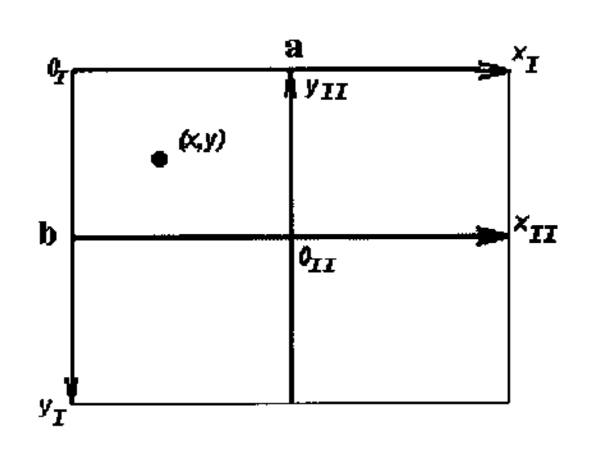


图 4－17　坐标系平移示意图

两种坐标系坐标变换矩阵表达式为：

$$\begin{bmatrix} x_{II} \\ y_{II} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -a \\ 0 & -1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix}$$

其逆变换转换矩阵表达式为：

$$\begin{bmatrix} x_I \\ y_I \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & -1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_{II} \\ y_{II} \\ 1 \end{bmatrix}$$

假设图像未旋转时中心坐标为（a, b），旋转后中心坐标为（c, d）（在新的坐标系下，以旋转后新图像左上角为原点），则旋转变换矩阵表达式为：

$$\begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & c \\ 0 & -1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1'_{II} \\ y1'_{II} \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & c \\ 0 & -1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1_{II} \\ y1_{II} \\ 1 \end{bmatrix}$$

$$= \begin{bmatrix} 1 & 0 & c \\ 0 & -1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -a \\ 0 & -1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix}$$

其逆变换矩阵表达式为：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & a \\ 0 & -1 & b \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -c \\ 0 & -1 & d \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x1 \\ y1 \\ 1 \end{bmatrix}$$

即：

$$\begin{bmatrix} x0 \\ y0 \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & -c\cos(\theta) - d\sin(\theta) + a \\ -\sin(\theta) & \cos(\theta) & c\sin(\theta) - d\cos(\theta) + b \\ 0 & 0 & 1 \end{bmatrix}$$

因此，

$$\begin{cases} x0 = x1\cos(\theta) + y1\sin(\theta) - c\cos(\theta) - d\sin(\theta) + a \\ y0 = -x1\sin(\theta) + y1\cos(\theta) + c\sin(\theta) - d\cos(\theta) + b \end{cases}$$

4.5.2 Visual C++编程实现

有了上面的转换公式，就可以非常方便的编写出实现图像旋转的函数。首先应计算出公式中需要的几个参数：a，b，c，d 和旋转后新图像的高、宽度。现在已知图像的原始宽度为 lWidth，高度为 lHeight，以图像中心为坐标系原点，则原始图像四个角的坐标分别为 $\left(-\frac{lWidth-1}{2}, \frac{lHeight-1}{2}\right)$，$\left(\frac{lWidth-1}{2}, \frac{lHeight-1}{2}\right)$，$\left(\frac{lWidth-1}{2}, -\frac{lHeight-1}{2}\right)$和$\left(-\frac{lWidth-1}{2}, -\frac{lHeight-1}{2}\right)$，按照旋转公式，在旋转后的新图中，这四个点坐标为：

$$(fDstX1, fDstY1) = \left(-\frac{lWidth-1}{2}\cos(\theta) + \frac{lHeight-1}{2}\sin(\theta), \frac{lWidth-1}{2}\sin(\theta) + \frac{lHeight-1}{2}\cos(\theta)\right)$$

$$(fDstX2, fDstY2) = \left(\frac{lWidth-1}{2}\cos(\theta) + \frac{lHeight-1}{2}\sin(\theta), -\frac{lWidth-1}{2}\sin(\theta) + \frac{lHeight-1}{2}\cos(\theta)\right)$$

$$(fDstX3, fDstY3) = \left(\frac{lWidth-1}{2}\cos(\theta) - \frac{lHeight-1}{2}\sin(\theta), -\frac{lWidth-1}{2}\sin(\theta) - \frac{lHeight-1}{2}\cos(\theta)\right)$$

$$(fDstX4, fDstY4) = \left(-\frac{lWidth-1}{2}\cos(\theta) - \frac{lHeight-1}{2}\sin(\theta), \frac{lWidth-1}{2}\sin(\theta) - \frac{lHeight-1}{2}\cos(\theta)\right)$$

则新图像的宽度 lNewWidth 和高度 lNewHeight 为：

$$lNewWidth = \max(|fDstX4 - fDstX1|, |fDstX3 - fDstX2|)$$

$$lNewHeight = \max(|fDstY4 - fDstY1|, |fDstY3 - fDstY2|)$$

令 $\begin{cases} f1 = -c\cos(\theta) - d\sin(\theta) + a \\ f2 = c\sin(\theta) - d\cos(\theta) + b \end{cases}$

因为 $a = \dfrac{lWidth-1}{2}, b = \dfrac{lHeight-1}{2}, c = \dfrac{lNewWidth-1}{2}, d = \dfrac{lNewHeight-1}{2}$

所以 $\begin{cases} f1 = -\dfrac{lNewWidth-1}{2}\cos(\theta) - \dfrac{lNewHeight-1}{2}\sin(\theta) + \dfrac{lWidth-1}{2} \\ f2 = \dfrac{lNewWidth-1}{2}\sin(\theta) - \dfrac{lNewHeight-1}{2}\cos(\theta) + \dfrac{lHeight-1}{2} \end{cases}$

则 $\begin{cases} x0 = x1\cos(\theta) + y1\sin(\theta) + f1 \\ y0 = -x1\sin(\theta) + y1\cos(\theta) + f2 \end{cases}$

下面就是实现图像旋转的函数 RotateDIB()的源代码。

```
/*************************************************************************
 *
 * 函数名称:
 *   RotateDIB()
 *
 * 参数:
 *   LPSTR lpDIB          - 指向原DIB的指针
 *   int iRotateAngle     - 旋转的角度（0-360度）
 *
 * 返回值:
 *   HGLOBAL              - 旋转成功返回新DIB句柄，否则返回NULL。
 *
 * 说明:
 *   该函数用来以图像中心为中心旋转DIB图像，返回新生成DIB的句柄。
 * 调用该函数会自动扩大图像以显示所有的像素。
 *
 ************************************************************************/

HGLOBAL WINAPI RotateDIB(LPSTR lpDIB, int iRotateAngle)
{

    // 原图像的宽度和高度
    LONG    lWidth;
    LONG    lHeight;

    // 旋转后图像的宽度和高度
    LONG    lNewWidth;
    LONG    lNewHeight;

    // 图像每行的字节数
```

```
LONG    lLineBytes;

// 旋转后图像的宽度（lNewWidth'，必须是4的倍数）
LONG    lNewLineBytes;

// 指向原图像的指针
LPSTR   lpDIBBits;

// 指向原像素的指针
LPSTR   lpSrc;

// 旋转后新DIB句柄
HDIB    hDIB;

// 指向旋转图像对应像素的指针
LPSTR   lpDst;

// 指向旋转图像的指针
LPSTR   lpNewDIB;
LPSTR   lpNewDIBBits;

// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFOHEADER lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREHEADER lpbmc;

// 循环变量（像素在新DIB中的坐标）
LONG    i;
LONG    j;

// 像素在原DIB中的坐标
LONG    i0;
LONG    j0;

// 旋转角度（弧度）
float   fRotateAngle;

// 旋转角度的正弦和余弦
float   fSina, fCosa;

// 原图四个角的坐标（以图像中心为坐标系原点）
float   fSrcX1,fSrcY1,fSrcX2,fSrcY2,fSrcX3,fSrcY3,fSrcX4,fSrcY4;

// 旋转后四个角的坐标（以图像中心为坐标系原点）
float   fDstX1,fDstY1,fDstX2,fDstY2,fDstX3,fDstY3,fDstX4,fDstY4;

// 两个中间常量
float   f1,f2;

// 找到原DIB图像像素起始位置
```

```
lpDIBBits = ::FindDIBBits(lpDIB);

// 获取图像的宽度
lWidth = ::DIBWidth(lpDIB);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 获取图像的高度
lHeight = ::DIBHeight(lpDIB);

// 将旋转角度从度转换到弧度
fRotateAngle = (float) RADIAN(iRotateAngle);

// 计算旋转角度的正弦
fSina = (float) sin((double)fRotateAngle);

// 计算旋转角度的余弦
fCosa = (float) cos((double)fRotateAngle);

// 计算原图的四个角的坐标（以图像中心为坐标系原点）
fSrcX1 = (float) (  (lWidth  - 1) / 2);
fSrcY1 = (float) (  (lHeight - 1) / 2);
fSrcX2 = (float) (  (lWidth  - 1) / 2);
fSrcY2 = (float) (  (lHeight - 1) / 2);
fSrcX3 = (float) (- (lWidth  - 1) / 2);
fSrcY3 = (float) (- (lHeight - 1) / 2);
fSrcX4 = (float) (  (lWidth  - 1) / 2);
fSrcY4 = (float) (  (lHeight - 1) / 2);

// 计算新图四个角的坐标（以图像中心为坐标系原点）
fDstX1 =  fCosa * fSrcX1 + fSina * fSrcY1;
fDstY1 = -fSina * fSrcX1 + fCosa * fSrcY1;
fDstX2 =  fCosa * fSrcX2 + fSina * fSrcY2;
fDstY2 = -fSina * fSrcX2 + fCosa * fSrcY2;
fDstX3 =  fCosa * fSrcX3 + fSina * fSrcY3;
fDstY3 = -fSina * fSrcX3 + fCosa * fSrcY3;
fDstX4 =  fCosa * fSrcX4 + fSina * fSrcY4;
fDstY4 = -fSina * fSrcX4 + fCosa * fSrcY4;

// 计算旋转后的图像实际宽度
lNewWidth  = (LONG) ( max( fabs(fDstX4 - fDstX1), fabs(fDstX3 - fDstX2) ) + 0.5);

// 计算新图像每行的字节数
lNewLineBytes = WIDTHBYTES(lNewWidth * 8);

// 计算旋转后的图像高度
lNewHeight = (LONG) ( max( fabs(fDstY4 - fDstY1), fabs(fDstY3 - fDstY2) )  + 0.5);

// 两个常数，这样不用以后每次都计算了
f1 = (float) (-0.5 * (lNewWidth - 1) * fCosa - 0.5 * (lNewHeight - 1) * fSina
```

```
    + 0.5 * (lWidth  - 1));
f2 = (float) ( 0.5 * (lNewWidth - 1) * fSina - 0.5 * (lNewHeight - 1) * fCosa
    + 0.5 * (lHeight - 1));

// 分配内存，以保存新DIB
hDIB = (HDIB) ::GlobalAlloc(GHND, lNewLineBytes * lNewHeight + *(LPDWORD)lpDIB
    + ::PaletteSize(lpDIB));

// 判断是否内存分配失败
if (hDIB == NULL)
{
    // 分配内存失败
    return NULL;
}

// 锁定内存
lpNewDIB =  (char * )::GlobalLock((HGLOBAL) hDIB);

// 复制DIB信息头和调色板
memcpy(lpNewDIB, lpDIB, *(LPDWORD)lpDIB + ::PaletteSize(lpDIB));

// 找到新DIB像素起始位置
lpNewDIBBits = ::FindDIBBits(lpNewDIB);

// 获取指针
lpbmi = (LPBITMAPINFOHEADER)lpNewDIB;
lpbmc = (LPBITMAPCOREHEADER)lpNewDIB;

// 更新DIB中图像的高度和宽度
if (IS_WIN30_DIB(lpNewDIB))
{
    // 对于Windows 3.0 DIB
    lpbmi->biWidth = lNewWidth;
    lpbmi->biHeight = lNewHeight;
}
else
{
    // 对于其他格式的DIB
    lpbmc->bcWidth = (unsigned short) lNewWidth;
    lpbmc->bcHeight = (unsigned short) lNewHeight;
}

// 针对图像每行进行操作
for(i = 0; i < lNewHeight; i++)
{
    // 针对图像每列进行操作
    for(j = 0; j < lNewWidth; j++)
    {
        // 指向新DIB第i行，第j个像素的指针
        // 注意此处宽度和高度是新DIB的宽度和高度
        lpDst = (char *)lpNewDIBBits + lNewLineBytes * (lNewHeight - 1 - i) + j;
```

```
			// 计算该像素在原DIB中的坐标
			i0 = (LONG) (-((float) j) * fSina + ((float) i) * fCosa + f2 + 0.5);
			j0 = (LONG) ( ((float) j) * fCosa + ((float) i) * fSina + f1 + 0.5);

			// 判断是否在原图范围内
			if( (j0 >= 0) && (j0 < lWidth) && (i0 >= 0) && (i0 < lHeight))
			{
				// 指向原DIB第i0行，第j0个像素的指针
				lpSrc = (char *)lpDIBBits + lLineBytes * (lHeight - 1 - i0) + j0;

				// 复制像素
				*lpDst = *lpSrc;
			}
			else
			{
				// 对于原图中没有的像素，直接赋值为255
				* ((unsigned char*)lpDst) = 255;
			}

		}

	}

	// 返回
	return hDIB;
}
```

在 CCh1_1View 中添加相应菜单事件处理代码：

```
void CCh1_1View::OnGeomRota()
{
	// 图像旋转

	// 获取文档
	CCh1_1Doc* pDoc = GetDocument();

	// 指向DIB的指针
	LPSTR lpDIB;

	// 锁定DIB
	lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

	// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的旋转，其他的可以类推）
	if (::DIBNumColors(lpDIB) != 256)
	{
		// 提示用户
		MessageBox("目前只支持256色位图的旋转！", "系统提示", MB_ICONINFORMATION | MB_OK);

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
```

```
    // 返回
    return;
}

// 缩放比率
int iRotateAngle;

// 创建对话框
CDlgGeoRota dlgPara;

// 初始化变量值
dlgPara.m_iRotateAngle = 90;

// 显示对话框，提示用户设定旋转角度
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户设定的旋转角度
iRotateAngle = dlgPara.m_iRotateAngle;

// 删除对话框
delete dlgPara;


// 创建新DIB
HDIB hNewDIB = NULL;

// 更改光标形状
BeginWaitCursor();

// 调用RotateDIB()函数旋转DIB
hNewDIB = (HDIB) ::RotateDIB(lpDIB, iRotateAngle);

// 判断旋转是否成功
if (hNewDIB != NULL)
{

    // 替换DIB，同时释放旧DIB对象
    pDoc->ReplaceHDIB(hNewDIB);

    // 更新DIB大小和调色板
    pDoc->InitDIBData();

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 重新设置滚动视图大小
    SetScrollSizes(MM_TEXT, pDoc->GetDocSize());
```

```
        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

上述代码中创建了一个 CdlgGeoRota dlgPara 对话框，它主要用来让用户设置旋转角度。图 4－2 旋转 30°后的结果如图 4－18 所示：

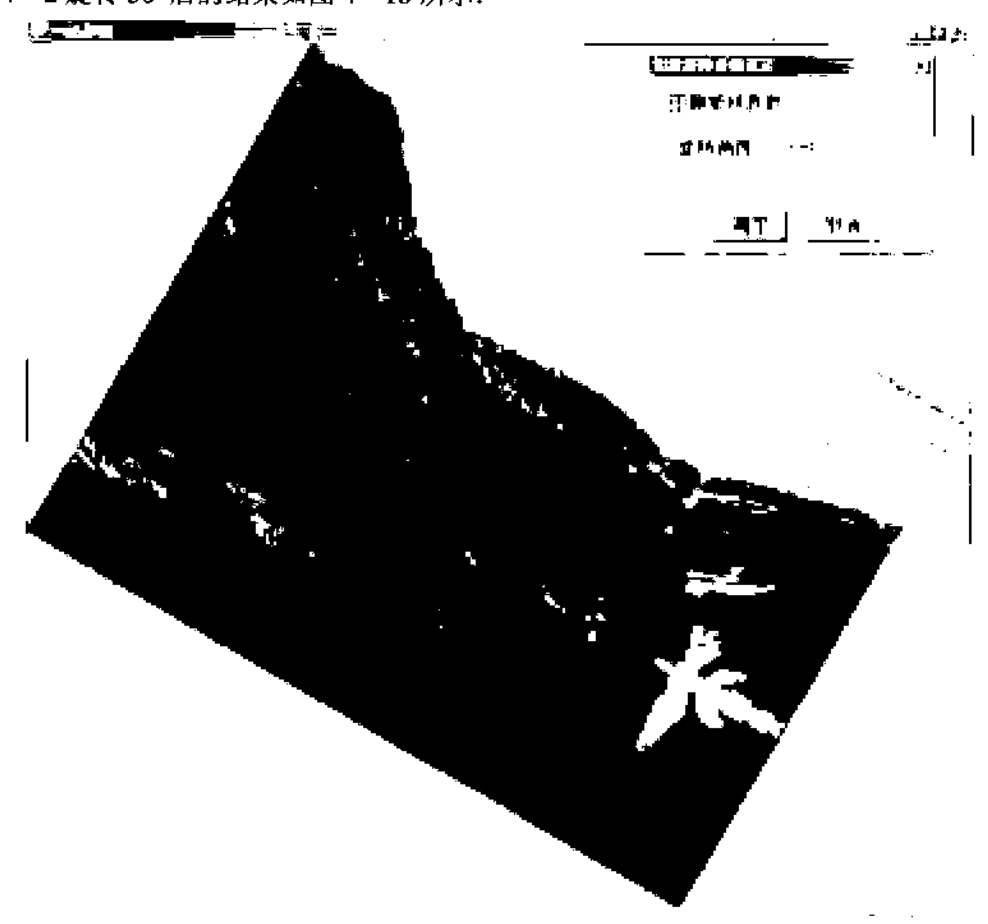


图 4－18　图像的缩放镜像

4.6 插值算法简介

前面已经提到，在对图像进行变换时可能产生一些原图中非整数位置的点，这时需要进行插值运算来计算出该点的像素值。下面详细介绍几种常用的插值算法。

4.6.1 最邻近插值

最邻近插值是一种简单的插值算法，也称为零阶插值。它输出的像素灰度值就等于距离它映射到的位置最近的输入像素的灰度值。最邻近插值算法最简单，上面代码中采用的插值算法都是它。然而，当图像中包含像素之间灰度级有变化的细微结构时，最近邻插值法会在图像中产生人为加工的痕迹。图 4—19 所示为一个用最近邻插值法旋转矩形图像的例子，结果图像带有锯齿形的边。

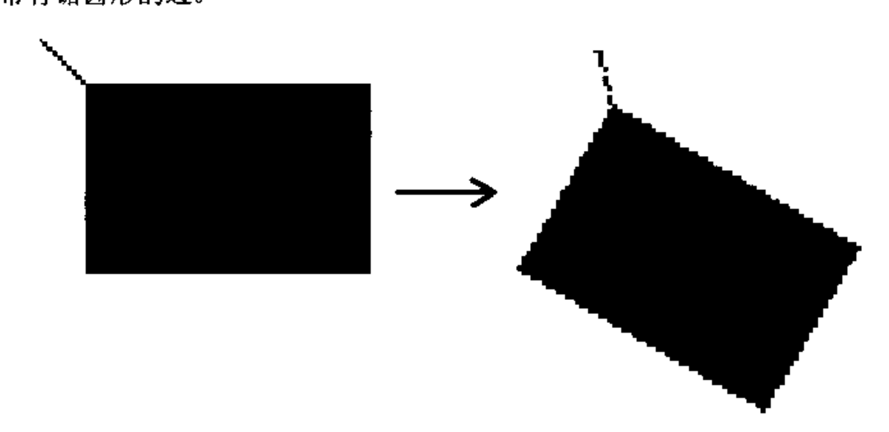


图 4－19 最近邻插值法旋转图像时产生的锯齿形边

4.6.2 双线性插值

双线性插值算法又称一阶插值算法，它的效果好于最邻近插值算法。只是程序相对复杂一些。运行时间稍长些。如图 4－20 所示，设 $0< x < 1$，$0< y < 1$。

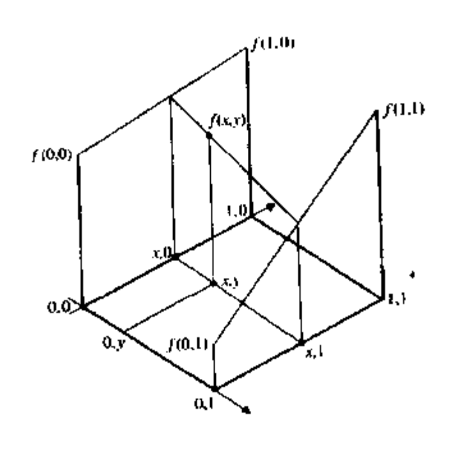


图 4－20 双线性插值示意图

首先可以通过一阶线性插值得出$f(x,0)$:

$$f(x,\ 0)=f(0,0)+x\,[f(1,0)-f(0,0)]$$

类似地，对$f(x,0)$进行一阶线性插值:

$$f(x,\ 1)=f(0,1)+x\,[f(1,1)-f(0,1)]$$

最后，对垂直方向进行一阶线性插值，以确定$f(x,\ y)$

$$f(x,\ y)=f(x,0)+y\,[f(x,1)-f(x,0)]$$

合并上述3式可得:

$$f(x,\ y)=[f(1,0)-f(0,0)]\,x+[f(0,1)-f(0,0)]\,y$$
$$+[f(1,1)+f(0,0)-f(0,1)-f(1,0)]xy+f(0,0)$$

一般情况下，在程序中进行双线性插值计算时直接用3次一阶线性插值即可。直接用3次一阶线性插值只要进行3次乘法和6次加减法运算，用上式只需要4次乘法和8次加减法运算。

上面的推导是在单位正方形上进行的，它可以推广到一般情况中使用。

下面我们利用双线性插值来更改一下旋转算法。

```
/*************************************************************************
 *
 * 函数名称:
 *   RotateDIB2()
 *
 * 参数:
 *   LPSTR lpDIB          - 指向原DIB的指针
 *   int iRotateAngle     - 旋转的角度（0-360度）
 *
 * 返回值:
 *   HGLOBAL              - 旋转成功返回新DIB句柄，否则返回NULL。
 *
 * 说明:
 *   该函数用来以图像中心为中心旋转DIB图像，返回新生成DIB的句柄。
 * 调用该函数会自动扩大图像以显示所有的像素。函数中采用双线性插
 * 值算法进行插值。
 *
 ************************************************************************/

HGLOBAL WINAPI RotateDIB2(LPSTR lpDIB, int iRotateAngle)
{

    // 原图像的宽度和高度
    LONG    lWidth;
    LONG    lHeight;

    // 旋转后图像的宽度和高度
    LONG    lNewWidth;
    LONG    lNewHeight;

    // 旋转后图像的宽度（lNewWidth'，必须是4的倍数）
    LONG    lNewLineBytes;
```

```
// 指向原图像的指针
LPSTR   lpDIBBits;

// 指向原像素的指针
LPSTR   lpSrc;

// 旋转后新DIB句柄
HDIB    hDIB;

// 指向旋转图像对应像素的指针
LPSTR   lpDst;

// 指向旋转图像的指针
LPSTR   lpNewDIB;
LPSTR   lpNewDIBBits;

// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFOHEADER lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREHEADER lpbmc;

// 循环变量（像素在新DIB中的坐标）
LONG    i;
LONG    j;

// 像素在原DIB中的坐标
FLOAT   i0;
FLOAT   j0;

// 旋转角度（弧度）
float   fRotateAngle;

// 旋转角度的正弦和余弦
float   fSina, fCosa;

// 原图四个角的坐标（以图像中心为坐标系原点）
float   fSrcX1,fSrcY1,fSrcX2,fSrcY2,fSrcX3,fSrcY3,fSrcX4,fSrcY4;

// 旋转后四个角的坐标（以图像中心为坐标系原点）
float   fDstX1,fDstY1,fDstX2,fDstY2,fDstX3,fDstY3,fDstX4,fDstY4;

// 两个中间常量
float   f1,f2;

// 找到原DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 获取图像的宽度
lWidth =::DIBWidth(lpDIB);
```

```
// 获取图像的高度
lHeight = ::DIBHeight(lpDIB);

// 将旋转角度从度转换到弧度
fRotateAngle = (float) RADIAN(iRotateAngle);

// 计算旋转角度的正弦
fSina = (float) sin((double)fRotateAngle);

// 计算旋转角度的余弦
fCosa = (float) cos((double)fRotateAngle);

// 计算原图的四个角的坐标(以图像中心为坐标系原点)
fSrcX1 = (float) (- (lWidth  - 1) / 2);
fSrcY1 = (float) (  (lHeight - 1) / 2);
fSrcX2 = (float) (  (lWidth  - 1) / 2);
fSrcY2 = (float) (  (lHeight - 1) / 2);
fSrcX3 = (float) (- (lWidth  - 1) / 2);
fSrcY3 = (float) (- (lHeight - 1) / 2);
fSrcX4 = (float) (  (lWidth  - 1) / 2);
fSrcY4 = (float) (- (lHeight - 1) / 2);

// 计算新图四个角的坐标(以图像中心为坐标系原点)
fDstX1 =  fCosa * fSrcX1 + fSina * fSrcY1;
fDstY1 = -fSina * fSrcX1 + fCosa * fSrcY1;
fDstX2 =  fCosa * fSrcX2 + fSina * fSrcY2;
fDstY2 = -fSina * fSrcX2 + fCosa * fSrcY2;
fDstX3 =  fCosa * fSrcX3 + fSina * fSrcY3;
fDstY3 = -fSina * fSrcX3 + fCosa * fSrcY3;
fDstX4 =  fCosa * fSrcX4 + fSina * fSrcY4;
fDstY4 = -fSina * fSrcX4 + fCosa * fSrcY4;

// 计算旋转后的图像实际宽度
lNewWidth  = (LONG) ( max( fabs(fDstX4 - fDstX1), fabs(fDstX3 - fDstX2) ) + 0.5);
lNewLineBytes = WIDTHBYTES(lNewWidth * 8);

// 计算旋转后的图像高度
lNewHeight = (LONG) ( max( fabs(fDstY4 - fDstY1), fabs(fDstY3 - fDstY2) )  + 0.5);

// 两个常数，这样不用以后每次都计算了
f1 = (float) (-0.5 * (lNewWidth - 1) * fCosa - 0.5 * (lNewHeight - 1) * fSina
    + 0.5 * (lWidth  - 1));
f2 = (float) ( 0.5 * (lNewWidth - 1) * fSina - 0.5 * (lNewHeight - 1) * fCosa
    + 0.5 * (lHeight - 1));

// 分配内存，以保存新DIB
hDIB = (HDIB) ::GlobalAlloc(GHND, lNewLineBytes * lNewHeight + *(LPDWORD)lpDIB
        + ::PaletteSize(lpDIB));

// 判断是否内存分配失败
```

```
if (hDIB == NULL)
{
    // 分配内存失败
    return NULL;
}

// 锁定内存
lpNewDIB =  (char * )::GlobalLock((HGLOBAL) hDIB);

// 复制DIB信息头和调色板
memcpy(lpNewDIB, lpDIB, *(LPDWORD)lpDIB + ::PaletteSize(lpDIB));

// 找到新DIB像素起始位置
lpNewDIBBits = ::FindDIBBits(lpNewDIB);

// 获取指针
lpbmi = (LPBITMAPINFOHEADER)lpNewDIB;
lpbmc = (LPBITMAPCOREHEADER)lpNewDIB;

// 更新DIB中图像的高度和宽度
if (IS_WIN30_DIB(lpNewDIB))
{
    // 对于Windows 3.0 DIB
    lpbmi->biWidth = lNewWidth;
    lpbmi->biHeight = lNewHeight;
}
else
{
    // 对于其他格式的DIB
    lpbmc->bcWidth = (unsigned short) lNewWidth;
    lpbmc->bcHeight = (unsigned short) lNewHeight;
}

// 针对图像每行进行操作
for(i = 0; i < lNewHeight; i++)
{
    // 针对图像每列进行操作
    for(j = 0; j < lNewWidth; j++)
    {
        // 指向新DIB第i行，第j个像素的指针
        // 注意此处宽度和高度是新DIB的宽度和高度
        lpDst = (char *)lpNewDIBBits + lNewLineBytes * (lNewHeight - 1 - i) + j;

        // 计算该像素在原DIB中的坐标
        i0 = -((float) j) * fSina + ((float) i) * fCosa + f2;
        j0 =  ((float) j) * fCosa + ((float) i) * fSina + f1;

        // 利用双线性插值算法来估算像素值
        *lpDst = Interpolation (lpDIBBits, lWidth, lHeight, j0, i0);

    }
```

```
    }

    // 返回
    return hDIB;
}


/*************************************************************************
 *
 * 函数名称:
 *   Interpolation()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *   FLOAT x            - 插值元素的x坐标
 *   FLOAT y            - 插值元素的y坐标
 *
 * 返回值:
 *   unsigned char      - 返回插值计算结果。
 *
 * 说明:
 *   该函数利用双线性插值算法来估算像素值。对于超出图像范围的像素，
 * 直接返回255。
 *
 ************************************************************************/

unsigned char WINAPI Interpolation (LPSTR lpDIBBits, LONG lWidth, LONG lHeight,
     FLOAT x, FLOAT y)
{

    // 四个最临近像素的坐标(i1, j1), (i2, j1), (i1, j2), (i2, j2)
    LONG    i1, i2;
    LONG    j1, j2;

    // 四个最临近像素值
    unsigned char   f1, f2, f3, f4;

    // 二个插值中间值
    unsigned char   f12, f34;

    // 定义一个值，当像素坐标相差小于改值时认为坐标相同
    FLOAT           EXP;

    // 图像每行的字节数
    LONG lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);
```

```
// 赋值
EXP = (FLOAT) 0.0001;

// 计算四个最临近像素的坐标
i1 = (LONG) x;
i2 = i1 + 1;
j1 = (LONG) y;
j2 = j1 + 1;

// 根据不同情况分别处理
if( (x < 0) || (x > lWidth - 1) || (y < 0) || (y > lHeight - 1))
{
    // 要计算的点不在原图范围内，直接返回255。
    return 255;
}
else
{
    if (fabs(x - lWidth + 1) <= EXP)
    {
        // 要计算的点在图像右边缘上
        if (fabs(y - lHeight + 1) <= EXP)
        {
            // 要计算的点正好是图像最右下角那一个像素，直接返回该点像素值
            f1 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i1);
            return f1;
        }
        else
        {
            // 在图像右边缘上且不是最后一点，直接一次插值即可
            f1 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i1);
            f3 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i2);

            // 返回插值结果
            return ((unsigned char) (f1 + (y -j1) * (f3 - f1)));
        }
    }
    else if (fabs(y - lHeight + 1) <= EXP)
    {
        // 要计算的点在图像下边缘上且不是最后一点，直接一次插值即可
        f1 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i1);
        f2 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j2) + i1);

        // 返回插值结果
        return ((unsigned char) (f1 + (x -i1) * (f2 - f1)));
    }
    else
    {
        // 计算四个最临近像素值
        f1 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i1);
        f2 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j2) + i1);
```

```
            f3 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j1) + i2);
            f4 = *((unsigned char *)lpDIBBits + lLineBytes * (lHeight - 1 - j2) + i2);

            // 插值1
            f12 = (unsigned char) (f1 + (x - i1) * (f2 - f1));

            // 插值2
            f34 = (unsigned char) (f3 + (x - i1) * (f4 - f3));

            // 插值3
            return ((unsigned char) (f12 + (y -j1) * (f34 - f12)));
        }
    }
}
```

如图 4－21 所示，左图是利用双线性插值算法进行旋转产生的效果；右图是利用最临近插值算法进行旋转产生的效果。可见，利用双线性插值算法进行插值过渡更加平滑。

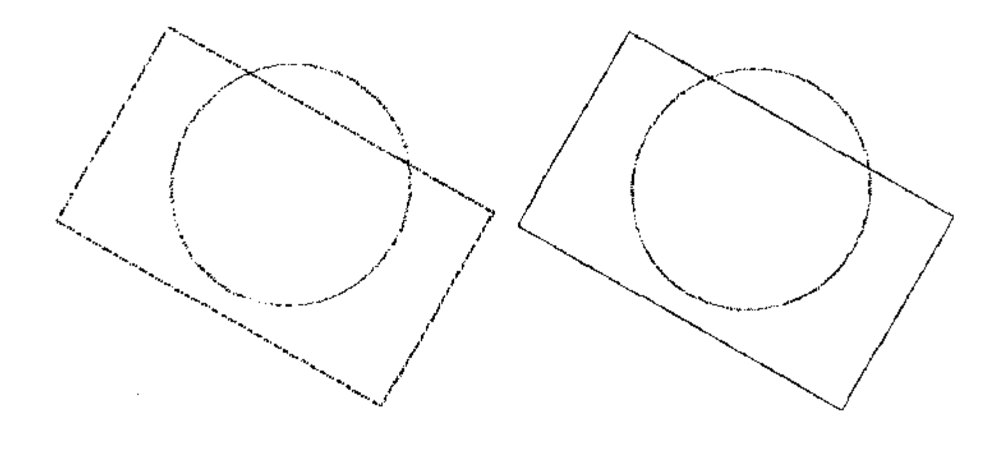


图 4－21　双线性插值和最临近插值效果对比

4.6.3　高阶插值

在几何运算中，双线性灰度插值的平滑作用可能会使图像的细节产生退化，在进行放大处理时，这种影响将更为明显。而在其他应用中，双线性插值的斜率不连续性会产生不希望的结果。这两种情况都可通过高阶插值得到修正，当然这需要增加计算量，这里就不详细介绍了。有兴趣的读者可以自己查看数值分析方面的教材。

第五章 图像的正交变换

数字图像处理的方法主要有两类：空间域处理法（空域法）及频域法（或者称变换域法）。前面几章介绍的点运算、几何变换等都是在空间域中进行图像处理的，本章中将介绍数字图像处理的一些常见的频域处理方法。

频域法首先是要将图像变换到频域，然后再进行处理。一般采用的变换方式都是线性正交变换，又称为酉变换。目前，图像的正交变换被广泛地运用于图像特征提取、图像增强、图像复原、图像压缩和图像识别等领域。

5.1 傅立叶变换

傅立叶变换是一种常见的正交变换，它在一维信号处理中得到了广泛的应用。在这里我们将它推广到数字图像处理中。

5.1.1 傅立叶变换的基本概念

傅立叶变换在数学中的定义是非常严格的，它的定义如下：

设 $f(x)$ 为 x 的函数，如果 $f(x)$ 满足下面的狄里赫莱条件：

(1) 具有有限个间隔点；
(2) 具有有限个极点；
(3) 绝对可积。

则定义 $f(x)$ 的傅立叶变换公式为：

$$F(\mu)=\int_{-\infty}^{+\infty} f(x)\mathrm{e}^{-\mathrm{j}2\pi\mu x}\mathrm{d}x \tag{5-1}$$

它的逆变换公式为：

$$f(x)=\int_{-\infty}^{+\infty} F(\mu)\mathrm{e}^{\mathrm{j}2\pi\mu x}\mathrm{d}\mu \tag{5-2}$$

其中 x 为时域变量，μ 为频域变量。如果再另 $\omega=2\pi\mu$，则上面两式可以写成：

$$F(\omega)=\int_{-\infty}^{+\infty} f(x)\mathrm{e}^{-\mathrm{j}\omega x}\mathrm{d}x \tag{5-3}$$

$$f(x)=\int_{-\infty}^{+\infty} F(\mu)\mathrm{e}^{\mathrm{j}\omega x}\mathrm{d}\mu=\frac{1}{2\pi}\int_{\infty}^{+\infty} F(\mu)\mathrm{e}^{\mathrm{j}\omega x}\mathrm{d}\omega \tag{5-4}$$

由上面的公式可以看出，傅立叶变换结果是一个复数表达式。设 $F(\mu)$ 的实部为 $R(\omega)$，虚部为 $I(\omega)$，则：

$$F(\omega)=R(\omega)+\mathrm{j}I(\omega) \tag{5-5}$$

或者写成指数形式：

$$F(\omega)=\left|F(\omega)\right|\mathrm{e}^{\mathrm{j}\phi(\omega)} \tag{5-6}$$

其中：

$$\left|F(\omega)\right|=\sqrt{R^2(\omega)+I^2(\omega)} \tag{5-7}$$

$$\phi(\omega)=\arctan\frac{I(\omega)}{R(\omega)} \tag{5-8}$$

通常称 $\left|F(\omega)\right|$ 为 $f(x)$ 的傅立叶幅度谱，$\phi(\omega)$ 为 $f(x)$ 的相位谱。

傅立叶变换页也可以推广到二维情况。如果二维函数 $f(x,y)$ 满足狄里赫莱条件，那么它的二维傅立叶变换为：

$$F(\mu,v)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(x,y)\mathrm{e}^{-\mathrm{j}2\pi(\mu x+vy)}\mathrm{d}x\mathrm{d}y \tag{5-9}$$

$$f(x,y)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}F(\mu,v)\mathrm{e}^{\mathrm{j}2\pi(\mu x+vy)}\mathrm{d}\mu\mathrm{d}v \tag{5-10}$$

同样，二维傅立叶变换的幅度谱和相位谱为：

$$\left|F(\mu,v)\right|=\sqrt{R^2(\mu,v)+I^2(\mu,v)} \tag{5-11}$$

$$\phi(\mu,v)=\arctan\frac{I(\mu,v)}{R(\mu,v)} \tag{5-12}$$

可以定义：

$$E(\mu,v)=R^2(\mu,v)+I^2(\mu,v) \tag{5-13}$$

通常称 $E(\mu,v)$ 为能量谱。

5.1.2 傅立叶变换的性质

傅立叶变换有很多重要的性质，这些性质为运算处理提供了方便。下面列出二维傅立叶变换的一些重要性质。

1. 可分性

一个二维傅立叶变换可以用二次一维傅立叶变换来实现。推导如下：

$$\begin{aligned}F(\mu,\nu) &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)\mathrm{e}^{-\mathrm{j}2\pi(\mu x+\nu y)}\mathrm{d}x\mathrm{d}y \\ &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} f(x,y)\mathrm{e}^{-\mathrm{j}2\pi\mu x}\mathrm{e}^{-\mathrm{j}2\pi\nu y}\mathrm{d}x\mathrm{d}y \\ &= \int_{-\infty}^{+\infty}\left[\int_{-\infty}^{+\infty} f(x,y)\mathrm{e}^{-\mathrm{j}2\pi\mu x}\mathrm{d}x\right]\mathrm{e}^{-\mathrm{j}2\pi\nu y}\mathrm{d}y \\ &= \int_{-\infty}^{+\infty}\{\Im_x[f(x,y)]\}\mathrm{e}^{-\mathrm{j}2\pi\nu y}\mathrm{d}y \\ &= \Im_y\{\Im_x[f(x,y)]\}\end{aligned} \tag{5-14}$$

2. 线性

傅立叶变换是一个线性变换，即：

$$\Im[\alpha\cdot f(x,y)+\beta\cdot g(x,y)] = \alpha\cdot\Im[f(x,y)]+\beta\cdot\Im[g(x,y)] \tag{5-15}$$

3. 对称性

如果函数 $f(x,y)$ 的傅立叶变换为 $F(\mu,\nu)$，那么：

$$\Im[F(x,y)] = f(-\mu,-\nu) \tag{5-16}$$

4. 尺度变换特性

如果函数 $f(x,y)$ 的傅立叶变换为 $F(\mu,\nu)$，α和β 为两个标量，那么：

$$\Im[\alpha f(x,y)] = \alpha F(\mu,\nu) \tag{5-17}$$

$$\Im[f(\alpha x,\beta y)] = \frac{1}{|\alpha\beta|}F(\frac{\mu}{\alpha},\frac{\nu}{\beta}) \tag{5-18}$$

5. 平移特性

傅立叶变换的平移特性如下所示：

$$\Im[f(x-x_0,y-y_0)] = F(\mu,\nu)\mathrm{e}^{-\mathrm{j}2\pi(\mu x_0+\nu y_0)} \tag{5-19}$$

$$\Im[f(x,y)\mathrm{e}^{\mathrm{j}2\pi(\mu_0 x+\nu_0 y)}] = F(\mu-\mu_0,\nu-\nu_0) \tag{5-20}$$

6. 共轭性

如果函数 $f(x,y)$ 的傅立叶变换为 $F(\mu,\nu)$，$F^*(-\mu,-\nu)$ 为 $f(-x,-y)$ 傅立叶变换的共轭函数，那么：

$$F(\mu,\nu) = F^*(-\mu,-\nu) \tag{5-21}$$

7. 旋转特性

如果空间域函数旋转角度为θ_0，则在变换域中该函数的傅立叶变换函数也将旋转同样的角度。数学表达式如下：

$$\Im[f(r,\theta+\theta_0)]=F(k,\phi+\theta_0) \tag{5-22}$$

公式 5－17 中 $f(r,\theta)$ 和 $F(k,\phi)$ 为极坐标表达式，其中 $x=r\cos\theta$， $y=r\sin\theta$，$\mu=k\cos\phi$， $v=k\sin\phi$， $f(r,\theta)$ 的傅立叶变换结果为 $F(k,\phi)$。

8. 能量保持定理

能量保持定理也称帕斯维尔（Parseval）定理。该定理的数学描述如下：

$$\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}|f(x,y)|^2\mathrm{d}x\mathrm{d}y=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}|F(\mu,v)|^2\mathrm{d}\mu\mathrm{d}v \tag{5-23}$$

该公式表明傅立叶变换前后能量守恒。

9. 相关定理

如果 $f(x,y)$ 和 $g(x,y)$ 为两个二维时域函数，那么可以定义相关运算∘如下：

$$f(x,y)\circ g(x,y)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)g(x+\alpha,y+\beta)\mathrm{d}\alpha\mathrm{d}\beta \tag{5-24}$$

则：

$$\Im[f(x,y)\circ g(x,y)]=F(\mu,v)\cdot G^*(\mu,v) \tag{5-25}$$

$$\Im[f(x,y)\cdot g^*(x,y)]=F(\mu,v)\circ G(\mu,v) \tag{5-26}$$

其中 $F(\mu,v)$ 为函数 $f(x,y)$ 的傅立叶变换，$G(\mu,v)$ 为函数 $g(x,y)$ 的傅立叶变换；$G^*(\mu,v)$ 为 $G(\mu,v)$ 的共轭， $g^*(x,y)$ 为 $g(x,y)$ 的共轭。

10. 卷积定理

如果 $f(x,y)$ 和 $g(x,y)$ 为两个二维时域函数，那么可以定义卷积运算＊如下：

$$f(x,y)*g(x,y)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)g(x-\alpha,y-\beta)\mathrm{d}\alpha\mathrm{d}\beta \tag{5-27}$$

则：

$$\Im[f(x,y)*g(x,y)]=F(\mu,v)\cdot G(\mu,v) \tag{5-28}$$

$$\Im[f(x,y)\cdot g(x,y)]=F(\mu,v)*G(\mu,v) \tag{5-29}$$

其中 $F(\mu,v)$ 为函数 $f(x,y)$ 的傅立叶变换，$G(\mu,v)$ 为函数 $g(x,y)$ 的傅立叶变换。

5.1.3 离散傅立叶变换

为了在数字图像处理中应用傅立叶变换，必须引入离散傅立叶变换（DFT，Discrete Fourier Transform）的概念。它的数学定义如下：

如果 $f(x)$ 为一个长度为 N 的数字序列，则其离散傅立叶变换 $F(\mu)$ 为：

$$F(\mu)=\Im[f(x)]=\sum_{x=0}^{N-1}f(x)e^{-j\frac{2\pi\mu x}{N}} \tag{5-30}$$

离散傅立叶反变换为：

$$f(x)=\Im^{-1}[F(\mu)]=\frac{1}{N}\sum_{\mu=0}^{N-1}F(\mu)e^{j\frac{2\pi\mu x}{N}} \tag{5-31}$$

其中：$x=0,1,2,\cdots,N-1$

如果令 $W=e^{j\frac{2\pi}{N}}$，那么上述公式变成：

$$F(\mu)=\Im[f(x)]=\sum_{x=0}^{N-1}f(x)e^{-j\frac{2\pi\mu x}{N}}=\sum_{x=0}^{N-1}f(x)W^{-\mu x} \tag{5-32}$$

$$f(x)=\Im^{-1}[F(\mu)]=\frac{1}{N}\sum_{\mu=0}^{N-1}F(\mu)e^{j\frac{2\pi\mu x}{N}}=\frac{1}{N}\sum_{\mu=0}^{N-1}F(\mu)W^{\mu x} \tag{5-33}$$

公式 5－32 写成矩阵形式为：

$$\begin{bmatrix}F(0)\\F(1)\\\vdots\\F(N-1)\end{bmatrix}=\begin{bmatrix}W^0 & W^0 & W^0 & \cdots & W^0\\W^0 & W^{1\times1} & W^{2\times1} & \cdots & W^{(N-1)\times1}\\\vdots & \vdots & \vdots & & \vdots\\W^0 & W^{1\times(N-1)} & W^{2\times(N-1)} & \cdots & W^{(N-1)\times(N-1)}\end{bmatrix}\begin{bmatrix}f(0)\\f(1)\\\vdots\\f(N-1)\end{bmatrix} \tag{5-34}$$

公式 5－33 写成矩阵形式为：

$$\begin{bmatrix}f(0)\\f(1)\\\vdots\\f(N-1)\end{bmatrix}=\frac{1}{N}\begin{bmatrix}W^0 & W^0 & W^0 & \cdots & W^0\\W^0 & W^{-1\times1} & W^{-2\times1} & \cdots & W^{-(N-1)\times1}\\\vdots & \vdots & \vdots & & \vdots\\W^0 & W^{-1\times(N-1)} & W^{-2\times(N-1)} & \cdots & W^{-(N-1)\times(N-1)}\end{bmatrix}\begin{bmatrix}F(0)\\F(1)\\\vdots\\F(N-1)\end{bmatrix} \tag{5-35}$$

同理，二维离散函数 $f(x,y)$ 的傅立叶变换为：

$$F(\mu,\nu)=\Im[f(x,y)]=\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi(\frac{\mu x}{M}+\frac{\nu y}{N})} \tag{5-36}$$

傅立叶反变换为：

$$f(x,y)=\Im^{-1}[F(\mu,\nu)]=\frac{1}{MN}\sum_{\mu=0}^{M-1}\sum_{\nu=0}^{N-1}F(\mu,\nu)e^{j2\pi(\frac{\mu x}{M}+\frac{\nu y}{N})} \tag{5-37}$$

其中：$x=0,1,2,\cdots,M-1$
$y=0,1,2,\cdots,N-1$

在数字图像处理中，图像取样一般是方阵，即 $M=N$，则二维离散傅立叶变换公式为：

$$F(\mu,\nu)=\Im[f(x,y)]=\sum_{x=0}^{N-1}\sum_{y=0}^{N-1}f(x,y)e^{-j2\pi(\frac{\mu x+\nu y}{N})} \tag{5-38}$$

$$f(x,y)=\Im^{-1}[F(\mu,\nu)]=\frac{1}{N^2}\sum_{\mu=0}^{N-1}\sum_{\nu=0}^{N-1}F(\mu,\nu)e^{j2\pi(\frac{\mu x+\nu y}{N})} \tag{5-39}$$

5.1.4 离散傅立叶变换的性质

离散傅立叶变换和联系傅立叶变换的性质类似，下面就列出二维离散傅立叶变换的一些重要性质。

1. 可分性

一个二维离散傅立叶变换也可以用二次一维离散傅立叶变换来实现：

$$F(\mu,\nu)=\Im_y\{\Im_x[f(x,y)]\} \tag{5-40}$$

2. 线性

离散傅立叶变换也是一个线性变换。即：

$$\Im[\alpha\cdot f(x,y)+\beta\cdot g(x,y)]=\alpha\cdot\Im[f(x,y)]+\beta\cdot\Im[g(x,y)] \tag{5-41}$$

说明：如果两个离散函数长度不同，则需要将长度短的那个函数末端补零，使两个离散函数长度一致。

3. 对称性

如果离散函数 $f(x,y)$ 的离散傅立叶变换为 $F(\mu,\nu)$，那么：

$$\Im[F(x,y)]=MN\cdot f(-\mu,-\nu) \tag{5-42}$$

4. 比例变换特性

如果函数 $f(x,y)$ 的傅立叶变换为 $F(\mu,\nu)$，α和β 为两个标量，那么：

$$\Im[\alpha f(x,y)] = \alpha F(\mu,\nu) \tag{5-43}$$

$$\Im[f(\alpha x,\beta y)] = \frac{1}{|\alpha\beta|}F(\frac{\mu}{\alpha},\frac{\nu}{\beta}) \tag{5-44}$$

5. 平移特性

离散傅立叶变换的平移特性如下所示：

$$\Im[f(x-x_0,y-y_0)] = F(\mu,\nu)\mathrm{e}^{-\mathrm{j}2\pi(\frac{\mu x_0}{M}+\frac{\nu y_0}{N})} \tag{5-45}$$

$$\Im[f(x,y)\mathrm{e}^{\mathrm{j}2\pi(\frac{\mu_0 x}{M}+\frac{\nu_0 y}{N})}] = F(\mu-\mu_0,\nu-\nu_0) \tag{5-46}$$

上面的公式表明如果在空间域中图像平移到点（x_0, y_0）处，则其对应的傅立叶变换要乘上一个系数$\mathrm{e}^{-\mathrm{j}2\pi(\frac{\mu x_0}{M}+\frac{\nu y_0}{N})}$，即表明空间域中图像的平移对应于频域中的相移，其傅立叶的幅值不变（因为$\mathrm{e}^{-\mathrm{j}2\pi(\frac{\mu x_0}{M}+\frac{\nu y_0}{N})}$的幅值为 1）。

在数字图像处理中通常将傅立叶变换频谱的原点移动到矩阵$M\times N$的中心，以便能清楚地分析傅立叶变换谱的情况。这样只要设：

$$\begin{cases}\mu_0 = \dfrac{M}{2}\\ \nu_0 = \dfrac{N}{2}\end{cases}，则\mathrm{e}^{\mathrm{j}2\pi(\frac{\mu_0 x}{M}+\frac{\nu_0 y}{N})} = \mathrm{e}^{\mathrm{j}\pi(x+y)} = (-1)^{x+y}。$$

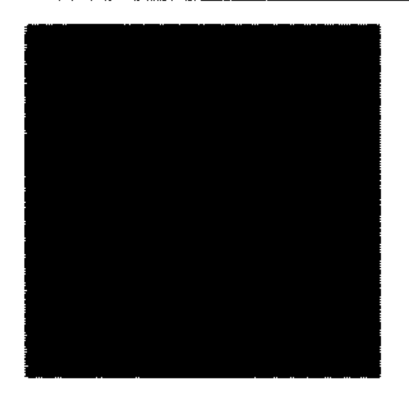
由公式 5－46 可得：

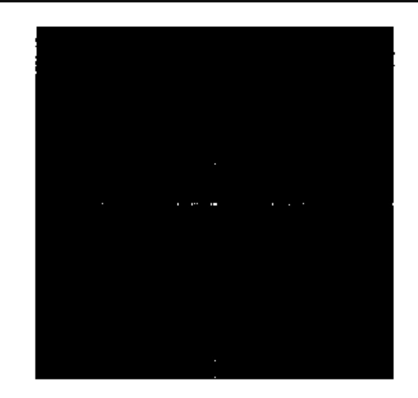
$$\Im[f(x,y)(-1)^{x+y}] = F(\mu-\frac{M}{2},\nu-\frac{N}{2}) \tag{5-47}$$

公式 5－47 表明，如果要将图像的频谱原点移动到图像中心，只要将$f(x,y)$乘上因子$(-1)^{x+y}$再进行离散傅立叶变换即可。图 5－1、图 5－2、图 5－3 为图像频谱移动示意图。



图 5－1　简单的方块图像


图 5－2　无平移地傅立叶频谱


图 5－3　平移后的傅立叶频谱

6. 周期性

离散傅立叶变换具有周期性：

$$F(\mu,v)=F(\mu+aN,v+bN) \tag{5-48}$$

$$f(x,y)=f(x+aN,y+bN) \tag{5-49}$$

其中 $a,b=0,\pm1,\pm2,\pm3,\cdots\cdots$

离散傅立叶变换的周期性说明离散函数 $f(x,y)$ 经过正变换后得到的 $F(\mu,v)$（或者 $F(\mu,v)$ 通过反变换得到的 $f(x,y)$）都是以 N 为周期的离散函数。

7. 共轭性

如果离散函数 $f(x,y)$ 的傅立叶变换为 $F(\mu,v)$，$F^*(-\mu,-v)$ 为 $f(-x,-y)$ 离散傅立叶变换的共轭函数，那么：

$$F(\mu,v)=F^*(-\mu,-v) \tag{5-50}$$

$$|F(\mu,v)|=|F(-\mu,-v)| \tag{5-51}$$

离散傅立叶变换的共轭性说明离散函数 $f(x,y)$ 经过正变换后得到的 $F(\mu,v)$ 是以原点为中心对称的。利用该特性，只要求出半个周期内的值就可以得到整个周期的值。

8. 旋转特性

如果空间域离散函数旋转角度 θ_0，则在变换域中该离散函数的离散傅立叶变换函数也将旋转同样的角度。数学表达式如下：

$$\Im[f(r,\theta+\theta_0)]=F(k,\phi+\theta_0) \tag{5-52}$$

公式 5－17 中 $f(r,\theta)$ 和 $F(k,\phi)$ 为离散函数极坐标表达式，其中 $x=r\cos\theta$，$y=r\sin\theta$，$\mu=k\cos\phi$，$v=k\sin\phi$，$f(r,\theta)$ 的离散傅立叶变换结果为 $F(k,\phi)$。

离散傅立叶变换的旋转特性如图 5－4 和图 5－5 所示。

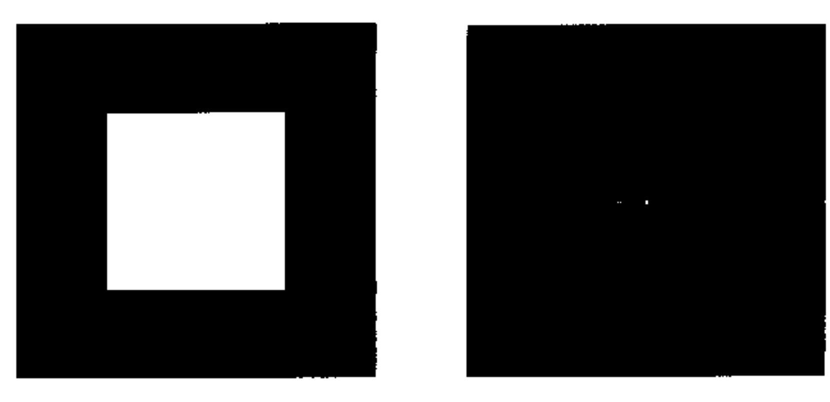
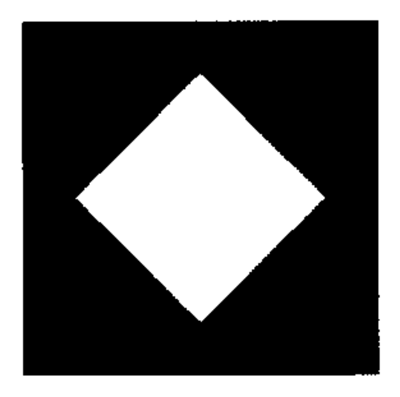


图 5－4 原始图像及其傅立叶频谱



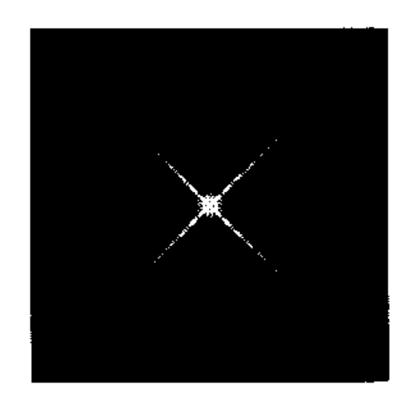


图 5－5 旋转 45 度后的图像及其傅立叶频谱

9. 平均值

如果定义离散图像 $f(x,y)$ 的平均值为：

$$\overline{f(x,y)}=\frac{1}{MN}\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}f(x,y) \tag{5－53}$$

那么：

$$\overline{f(x,y)}=F(0,0) \tag{5－54}$$

即：二维离散图像的平均值即为其离散傅立叶变换在原点的值。

10. 能量保持定理

能量保持定理也称帕斯维尔（Parseval）定理。该定理的数学描述如下：

$$\sum_{x=0}^{M-1}\sum_{y=0}^{N-1}\left|f(x,y)\right|^2=\sum_{\mu=0}^{M-1}\sum_{\nu=0}^{N-1}\left|F(\mu,\nu)\right|^2 \tag{5-55}$$

该公式表明离散傅立叶变换前后能量守恒。

11. 微分特性

如果定义二维离散函数 $f(x,y)$ 的拉普拉斯算子为：

$$\nabla^2 f(x,y)=\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2}$$

那么下式成立：

$$\Im\left\{\nabla^2 f(x,y)\right\}=-(2\pi)^2(\mu^2+\nu^2)F(\mu,\nu)$$

12. 卷积定理

对于离散函数 $f(x,y)$ 和 $g(x,y)$ 的卷积运算 * 要稍微复杂一些，必须首先对 $f(x,y)$ 和 $g(x,y)$ 的定义域进行扩展，以防止卷积后产生交叠误差（Wraparound Error）。

设 $f(x,y)$ 和 $g(x,y)$ 分别是 $A\times B$ 和 $C\times D$ 的二维离散数组，即 $f(x,y)$ 的定义域为：$x=0,1,\cdots,A-1$，$y=0,1,\cdots,B-1$，$g(x,y)$ 的定义域为：$x=0,1,\cdots,C-1$，$y=0,1,\cdots,D-1$。为了防止交叠误差，必须假设这些数组在 x 和 y 方向延伸为周期为 M 和 N 的周期函数，其中 M 和 N 的大小为：

$$\begin{cases}M\ge A+C-1\\ N\ge B+D-1\end{cases}$$

将 $f(x,y)$ 和 $g(x,y)$ 用增补零的办法进行扩充：

$$f_e(x,y)=\begin{cases}f(x,y) & 0\le x\le A-1,\quad 0\le y\le B-1\\ 0 & A\le x\le M-1,\quad B\le y\le N-1\end{cases}$$

$$g_e(x,y)=\begin{cases}g(x,y) & 0\le x\le C-1,\quad 0\le y\le D-1\\ 0 & C\le x\le M-1,\quad D\le y\le N-1\end{cases}$$

其二维离散卷积定义为：

$$f_e(x,y)*g_e(x,y)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f_e(m,n)g_e(x-m,y-n) \tag{5-56}$$

则二维离散傅立叶卷积定理如下：

$$\Im[f_e(x,y)*g_e(x,y)]=F_e(\mu,v)\cdot G_e(\mu,v) \tag{5-57}$$

$$\Im[f_e(x,y)\cdot g_e(x,y)]=F_e(\mu,v)*G_e(\mu,v) \tag{5-58}$$

其中 $F_e(\mu,v)$ 为离散函数 $f_e(x,y)$ 的离散傅立叶变换；$G_e(\mu,v)$ 为离散函数 $g_e(x,y)$ 的离散傅立叶变换。上面的公式和连续情况的基本相同，只是所有的变量 x,y,μ,v 都是离散变量，而且运算是针对扩充函数 $f_e(x,y)$ 和 $g_e(x,y)$ 进行的。

13. 相关定理

对于离散情况下的相关定理，同样先必须对 $f(x,y)$ 和 $g(x,y)$ 进行扩充处理，然后再定义相关运算○如下：

$$f_e(x,y)\circ g_e(x,y)=\sum_{m=0}^{M-1}\sum_{n=0}^{N-1}f_e(m,n)g_e(x+m,y+n) \tag{5-59}$$

则：

$$\Im[f_e(x,y)\circ g_e(x,y)]=F_e(\mu,v)\cdot G_e^*(\mu,v) \tag{5-60}$$

$$\Im[f_e(x,y)\cdot g_e^*(x,y)]=F_e(\mu,v)\circ G_e(\mu,v) \tag{5-61}$$

其中 $F_e(\mu,v)$ 为离散函数 $f_e(x,y)$ 的离散傅立叶变换；$G_e(\mu,v)$ 为离散函数 $g_e(x,y)$ 的离散傅立叶变换；$G_e^*(\mu,v)$ 为 $G_e(\mu,v)$ 的共轭，$g_e^*(x,y)$ 为 $g_e(x,y)$ 的共轭。

5.1.5 快速傅立叶变换

随着计算机技术和数字电路的迅速发展，离散傅立叶变换已经成为数字信号处理和图像处理的一种重要的手段。然而，离散傅立叶变换需要的计算量太大，运算时间长，在某种程度上限制了它的使用。按照公式 5—30，计算一个长度为 N 的一维离散傅立叶变换，对 μ 的每一个值需要做 N 次复数乘法和（$N-1$）次复数加法。那么对 N 个 μ，则需要 N^2 次复数乘法和 $N(N-1)\approx N^2$ 次复数加法。很显然，当 N 很大时，计算量是相当可观的。

1965 年，库里（Cooley）和图基（Tukey）首先提出一种快速傅立叶变换（FFT）算法，采用该算法进行离散傅立叶变换，复数乘法和加法次数正比于 $N\log_2 N$，这在 N 很大时计算量会大大减少。如表 5－1 所示：

表 5－1　　FFT 算法与普通傅立叶变换算法的对比

N	N^2（普通 FT）	$N\log_2 N$ （FFT）	计算收益（$N/\log_2 N$）
2	4	2	2.0
4	16	8	2.0
8	64	24	2.7
16	256	64	4.0
32	1024	160	6.4
64	4096	384	10.7
128	16384	896	18.3
256	65536	2048	32.0
512	262144	4608	56.9
1024	1048576	10240	102.4
2048	4194304	22528	186.2

可见，采用 FFT 可以减少运算量，图像越大减少越多。对于长为 1024 的离散序列，用普通的离散傅立叶变换往往要计算几十分钟，而采用 FFT，一般只要几十秒。

快速傅立叶变换不是一种新的变换，它只是离散傅立叶变换的一种改进算法。它分析了离散傅立叶变换中重复的计算量，并尽最大的可能使之减少，从而达到快速计算的目的。下面我们来从一维离散傅立叶变换来讨论。

一维离散傅立叶变换可以用矩阵表达式 5－34 来表示：

$$\begin{bmatrix} F(0) \\ F(1) \\ \vdots \\ F(N-1) \end{bmatrix} = \begin{bmatrix} W^0 & W^0 & W^0 & \cdots & W^0 \\ W^0 & W^{1\times1} & W^{2\times1} & \cdots & W^{(N-1)\times1} \\ \vdots & \vdots & \vdots & & \vdots \\ W^0 & W^{1\times(N-1)} & W^{2\times(N-1)} & \cdots & W^{(N-1)\times(N-1)} \end{bmatrix} \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{bmatrix}$$

观察系数矩阵，并结合 W 的定义表达式 $W = e^{j\frac{2\pi}{N}}$，可以发现系数 $W^{m\times n}$ 是以 N 为周期。这样系数矩阵中很多系数是相同的，不必进行多次计算。而且由于 $W^{\frac{N}{2}} = e^{j\frac{2\pi}{N}\times\frac{N}{2}} = -1$，因此 $W^{m\times n+\frac{N}{2}} = W^{m\times n} \times W^{\frac{N}{2}} = -W^{m\times n}$，即 $W^{m\times n}$ 又具有对称性，因而利用 $W^{m\times n}$ 的对称性可以进一步减少计算量。

例如，对于 N＝4，系数矩阵为：

$$\begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & W^2 & W^3 \\ W^0 & W^2 & W^4 & W^6 \\ W^0 & W^3 & W^6 & W^9 \end{bmatrix}$$

由 $W^{m\times n}$ 的周期性可以得出：$W^4 = W^0$，$W^6 = W^2$，$W^9 = W^1$；由 $W^{m\times n}$ 的对称性可

以得出：$W^3=-W^1$，$W^2=-W^0$。因此系数矩阵可以转化为：

$$\begin{bmatrix} W^0 & W^0 & W^0 & W^0 \\ W^0 & W^1 & -W^0 & -W^1 \\ W^0 & -W^0 & W^0 & -W^0 \\ W^0 & -W^1 & -W^0 & W^1 \end{bmatrix}$$

可见系数矩阵的系数重复是很多的，如果把序列分解成若干短序列，并与系数矩阵元素巧妙的结合起来计算离散傅立叶变换，可以简化运算，这就是 FFT 的基本思路。

设 $N=2^\beta$（β 为整数），下面我们按照奇偶来将序列 $x(n)$ 进行划分，设：

$$\begin{cases} g(n)=x(2n) \\ h(n)=x(2n+1) \end{cases} \qquad (n=0,1,2,3,\cdots,\frac{N}{2}-1)$$

因此，离散傅立叶变换可以改写成下面的形式：

$$\begin{aligned} X(m) &= \sum_{n=0}^{N-1} x(n)\cdot W_N^{mn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} g(n)\cdot W_N^{mn} + \sum_{n=0}^{\frac{N}{2}-1} h(n)\cdot W_N^{mn} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)\cdot W_N^{m(2n)} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\cdot W_N^{m(2n+1)} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n)\cdot W_{\frac{N}{2}}^{mn} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1)\cdot W_{\frac{N}{2}}^{mn}\cdot W_N^{m} \\ &= G(m)+W_N^m\cdot H(m) \end{aligned} \tag{5-62}$$

因此，一个求 N 点的 DFT 可以被转换成两个求 $\frac{N}{2}$ 点的 DFT。以 N＝8 的 DFT 为例，利用公式 5－62 可得：

$$\begin{cases} X(0)=G(0)+W_8^0\cdot H(0) \\ X(1)=G(1)+W_8^1\cdot H(1) \\ X(2)=G(2)+W_8^2\cdot H(2) \\ X(3)=G(3)+W_8^3\cdot H(3) \\ X(4)=G(4)+W_8^4\cdot H(4) \end{cases} \tag{5-63}$$

$$\begin{cases} X(5) = G(5) + W_8^5 \cdot H(5) \\ X(6) = G(6) + W_8^6 \cdot H(6) \\ X(7) = G(7) + W_8^7 \cdot H(7) \end{cases}$$

由于$G(m)$和$H(m)$都是4点的DFT，所以它们均以4为周期。因此$G(m+4) = G(m)$，$H(m+4) = H(m)$。再加上W_8^m的对称性$W_8^{m+4} = -W_8^m \quad (m = 0, 1, 2, 3)$，因此公式5－63可以改进为：

$$\begin{cases} X(0) = G(0) + W_8^0 \cdot H(0) \\ X(1) = G(1) + W_8^1 \cdot H(1) \\ X(2) = G(2) + W_8^2 \cdot H(2) \\ X(3) = G(3) + W_8^3 \cdot H(3) \\ X(4) = G(0) - W_8^0 \cdot H(0) \\ X(5) = G(1) - W_8^1 \cdot H(1) \\ X(6) = G(2) - W_8^2 \cdot H(2) \\ X(7) = G(3) - W_8^3 \cdot H(3) \end{cases} \tag{5-64}$$

如图5－6所示，定义由G(1)、H(1)、X(1)和X(5)所构成的结构为蝶形运算单元，它的左方两个节点为输入节点，代表输入数值；右方两个节点为输出节点，表示输入数值的叠加。运算由左向右进行。线旁的W_8^1和$-W_8^1$为加权系数，图5－6表示的运算为：

$$X(1) = G(1) + W_8^1 \cdot H(1)$$

$$X(5) = G(1) - W_8^1 \cdot H(1)$$

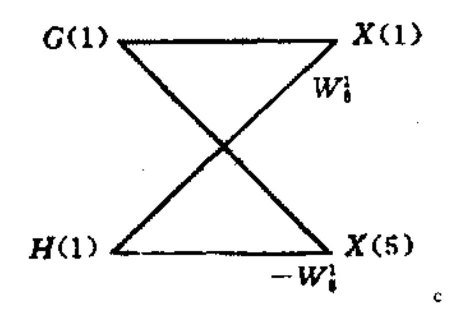


图5－6　蝶形运算单元

蝶形运算单元也可以如图5－7来表达：

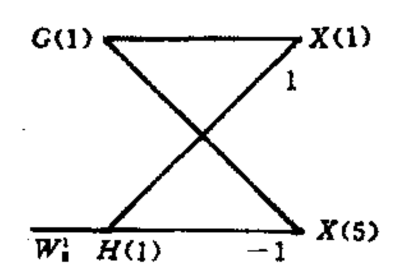


图 5−7 蝶形运算单元另外一种画法

这样，公式 5−64 就可以用蝶形运算单元来表示。如图 5−8 所示：

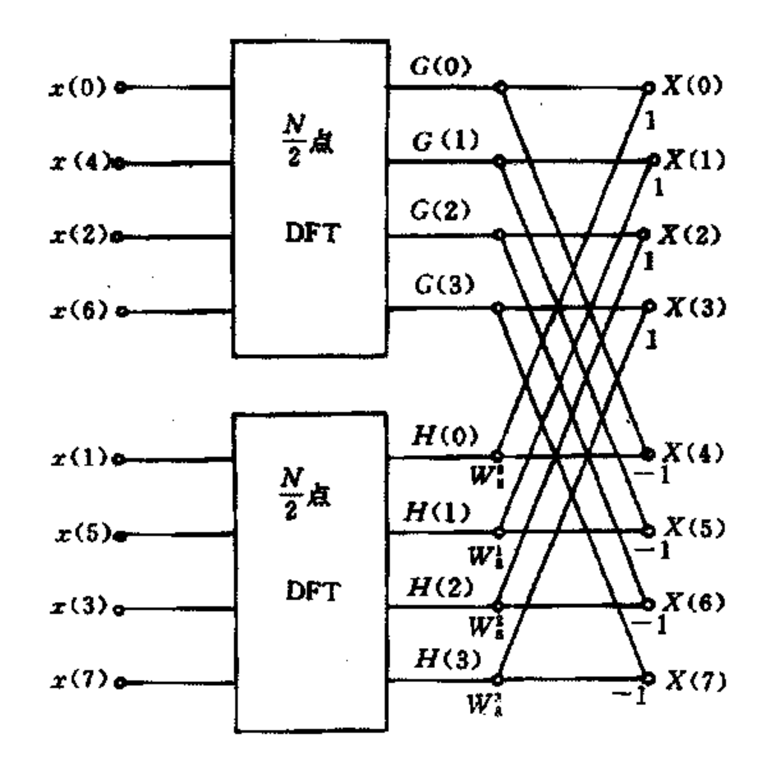


图 5−8 蝶形运算单元

$G(m)$ 和 $H(m)$ 都是 4 点的 DFT，如果对它们再按照奇偶进行分组：

$$\begin{cases} a(n) = g(2n) \\ b(n) = g(2n+1) \end{cases} \qquad (n = 0, 1, 2, 3, \cdots, \frac{N}{4} - 1)$$

$$\begin{cases} c(n) = h(2n) \\ d(n) = h(2n+1) \end{cases} \qquad (n = 0, 1, 2, 3, \cdots, \frac{N}{4} - 1)$$

则：

$$
\begin{aligned}
G(m) &= \sum_{n=0}^{\frac{N}{2}-1} g(n) \cdot W_{\frac{N}{2}}^{mn} \\
&= \sum_{n=0}^{\frac{N}{4}-1} g(2n) \cdot W_{\frac{N}{2}}^{m2n} + \sum_{n=0}^{\frac{N}{4}-1} g(2n+1) \cdot W_{\frac{N}{2}}^{m(2n+1)} \\
&= \sum_{n=0}^{\frac{N}{4}-1} a(n) \cdot W_{\frac{N}{4}}^{mn} + \sum_{n=0}^{\frac{N}{2}-1} b(n) \cdot W_{\frac{N}{4}}^{mn} \cdot W_{\frac{N}{2}}^{m} \\
&= A(m) + W_N^{2m} \cdot B(m)
\end{aligned}
\tag{5-65}
$$

同理：

$$
H(m) = C(m) + W_N^{2m} \cdot D(m) \tag{5-66}
$$

因设 N=8，故 $A(m)$、$B(m)$、$C(m)$ 和 $D(m)$ 都是 2 点的 DFT，它们与 $G(m)$ 和 $H(m)$ 的关系为：

$$
\begin{cases}
G(0) = A(0) + W_8^0 \cdot B(0) \\
G(1) = A(1) + W_8^2 \cdot B(1) \\
G(2) = A(0) - W_8^0 \cdot B(2) \\
G(3) = A(1) - W_8^2 \cdot B(3)
\end{cases}
\qquad
\begin{cases}
H(0) = C(0) + W_8^0 \cdot D(0) \\
H(1) = C(1) + W_8^2 \cdot D(1) \\
H(2) = C(0) - W_8^0 \cdot D(2) \\
H(3) = C(1) - W_8^2 \cdot D(3)
\end{cases}
$$

这样，由 $A(m)$、$B(m)$、$C(m)$ 和 $D(m)$ 计算 $G(m)$ 和 $H(m)$ 的蝶形流图如图 5－9 所示。

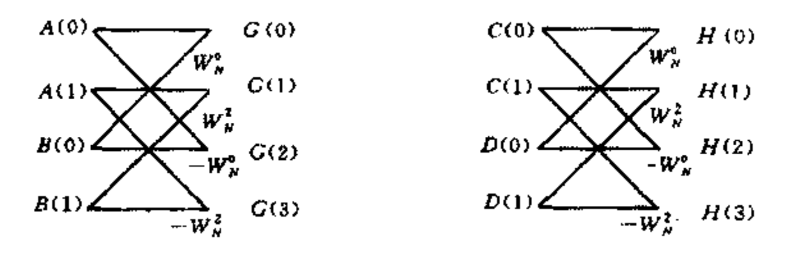


图 5－9　4 点 DFT 分解成 2 点 DFT 的蝶形流图

至此，$A(m)$、$B(m)$、$C(m)$ 和 $D(m)$ 都已经是 2 点的 DFT，它们可以由原始数据 $x(n)$ 直接求出。计算公式如下：

$$
\begin{cases}
A(0) = x(0) + W_8^0 \cdot x(4) \\
A(1) = x(0) - W_8^0 \cdot x(4)
\end{cases}
\qquad
\begin{cases}
B(0) = x(2) + W_8^0 \cdot x(6) \\
B(1) = x(2) - W_8^0 \cdot x(6)
\end{cases}
$$

$$\begin{cases} C(0) = x(1) + W_8^0 \cdot x(5) \\ C(1) = x(1) - W_8^0 \cdot x(5) \end{cases} \qquad \begin{cases} D(0) = x(3) + W_8^0 \cdot x(7) \\ D(1) = x(3) - W_8^0 \cdot x(7) \end{cases}$$

综上所述，8 点 DFT 的完整蝶形计算流图和逐级分解框图分别如图 5－10 和图 5－11 所示。由图 5－10 得出的算法即快速傅立叶变换（FFT）。

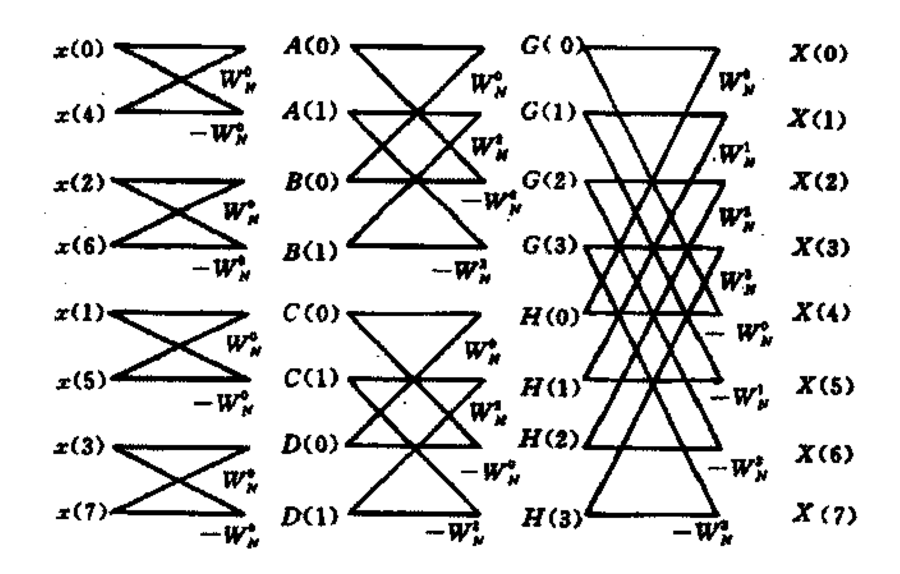


图 5－10 8 点 DFT 蝶形流图

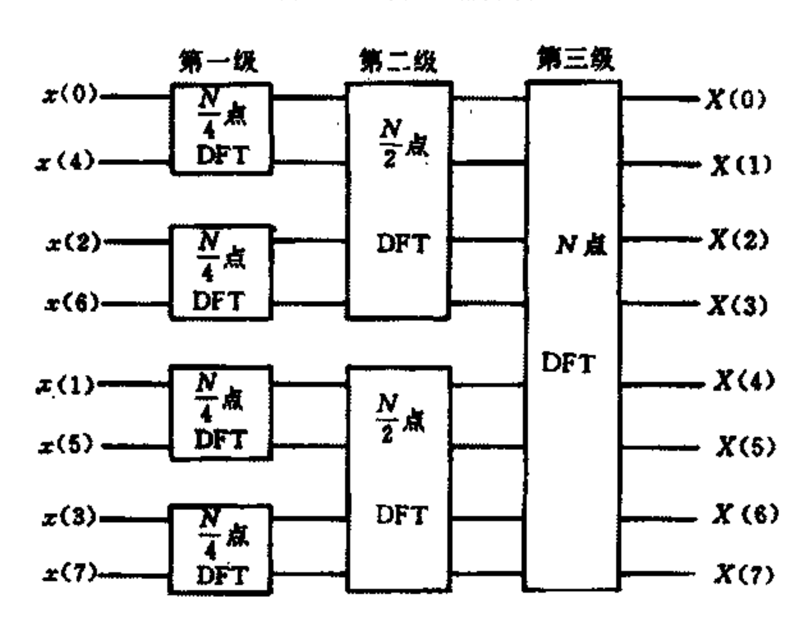


图 5－11 8 点 DFT 逐级分解运算框图

由图 5－10 可见，蝶形流图的输出序列 $X(m)$ 是按照 m 从小到大的顺序排列的，而输入

序列 $x(n)$ 不是从小到大的顺序，是按照所谓的码位倒序而排列的。

如果把自然顺序的十进制数转换成二进制数，然后将这些二进制的首末位倒序再重新转换成十进制数，那么这时的十进制数的排列就是码位倒序排列。N＝8 的自然顺序与码位倒序的比较如表 5－2 所示。

表 5－2　　自然顺序与码位倒序（N＝8）

十进制数	二进制数	二进制数的码位倒序	码位倒序后的十进制数
0	000	000	0
1	001	100	4
2	010	010	2
3	011	110	6
4	100	001	1
5	101	101	5
6	110	011	3
7	111	111	7

码位倒序的排列规律是由 FFT 算法所决定的，如果要求输入按照自然顺序排列，则输出就必然为码位倒序排列。如图 5－12 所示：

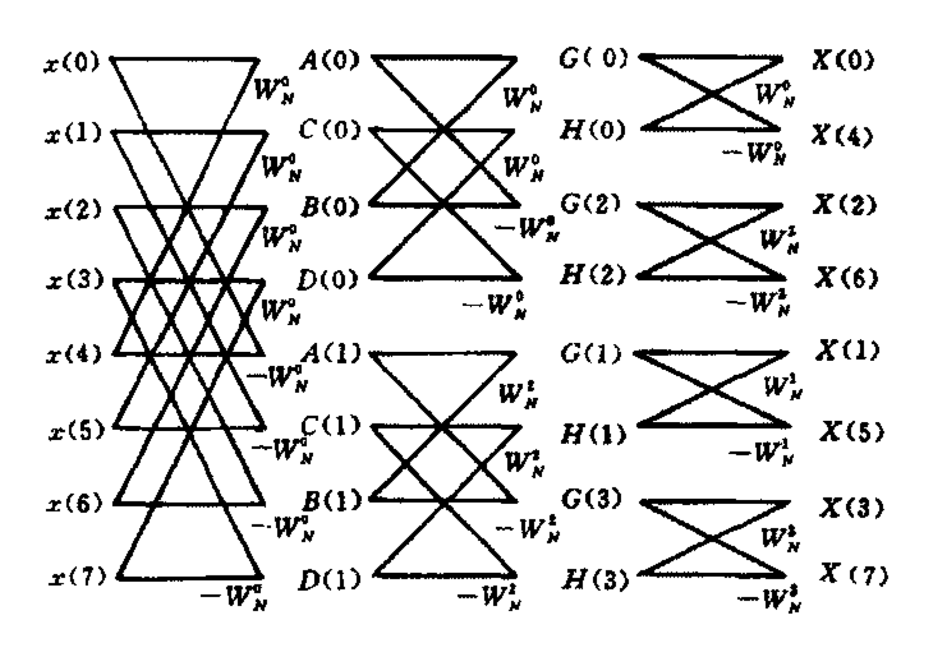


图 5－12　8 点 DFT 蝶形流图（输入为自然顺序）

由图 5－10 可以计算出 FFT 的运算量。每按奇偶分解一次即可得一级蝶形流图，当 N＝8 时，共有 $\log_2 8=3$ 级蝶形流图。又由于每级蝶形流图都有 $\frac{N}{2}=4$ 个蝶形单元，故三级共有蝶形单元 $\frac{N}{2}\log_2 N=4\times 3=12$（个）。每个蝶形单元需要一次复数乘法和二次复数加法运算，因此 8 点 FFT 一共需要复数乘法运算 $\frac{N}{2}\log_2 N=4\times 3=12$ 次，复数加法运算

$N\log_2 N = 24$次。而如果直接采用 DFT 的定义计算，8 点 DFT 一共需要复数乘法运算$N^2 = 64$次，复数加法运算$N(N-1) = 56$次。当N增大时，算法效率将急剧增加。

上面介绍的 FFT 算法是把时间序列$x(n)$按照n的奇偶进行分组计算的，又称为按时间分组的 FFT 算法。如果将频率序列$X(m)$按照m的奇偶进行分组分解在来计算，则称为按照频率分组的 FFT 算法。两者推导过程类似，计算量一样，仅仅是算法结构不同而已。如果N不是 2 的整数次幂，则它的 FFT 算法比较复杂，这里就不再介绍。有兴趣的读者可以查阅有关信号处理方面的书籍。

5.1.6 Visual C++编程实现图像傅立叶变换

有了上面的理论基础，我们就可以编写自己的 FFT 程序了。下面定义的函数 FFT()就可以完成一维离散快速傅立叶变换。它有三个参数：一个是指向时域数组的指针 TD，该数组保存着要进行傅立叶变换的数值序列，类型为复数；第二个是指向频域数组的指针，用来保存快速傅立叶变换结果。参数r为$\log_2 N$，即为迭代次数。傅立叶变换的点数可以由参数r直接求出，只要将 1 左移r位（就是2^r）即可。

下面给出函数 FFT()的完整代码。

```
// 常数π
#define PI 3.1415926535

/*************************************************************************
 *
 * 函数名称:
 *   FFT()
 *
 * 参数:
 *   complex<double> * TD   - 指向时域数组的指针
 *   complex<double> * FD   - 指向频域数组的指针
 *   r                      -2的幂数，即迭代次数
 *
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速傅立叶变换。
 *
 ************************************************************************/

VOID WINAPI FFT(complex<double> * TD, complex<double> * FD, int r)
{
    // 傅立叶变换点数
    LONG    count;

    // 循环变量
    int     i,j,k;

    // 中间变量
    int     bfsize,p;
```

```
// 角度
double  angle;

complex<double> *W,*X1,*X2,*X;

// 计算傅立叶变换点数
count = 1 << r;

// 分配运算所需存储器
W  = new complex<double>[count / 2];
X1 = new complex<double>[count];
X2 = new complex<double>[count];

// 计算加权系数
for(i = 0; i < count / 2; i++)
{
    angle = -i * PI * 2 / count;
    W[i] = complex<double> (cos(angle), sin(angle));
}

// 将时域点写入X1
memcpy(X1, TD, sizeof(complex<double>) * count);

// 采用蝶形算法进行快速傅立叶变换
for(k = 0; k < r; k++)
{
    for(j = 0; j < 1 << k; j++)
    {
        bfsize = 1 << (r-k);
        for(i = 0; i < bfsize / 2; i++)
        {

            p = j * bfsize;
            X2[i + p] = X1[i + p] + X1[i + p + bfsize / 2];
            X2[i + p + bfsize / 2] = (X1[i + p] - X1[i + p + bfsize / 2]) * W[i * (1<<k)];
        }
    }
    X  = X1;
    X1 = X2;
    X2 = X;
}

// 重新排序
for(j = 0; j < count; j++)
{
    p = 0;
    for(i = 0; i < r; i++)
    {
        if (j&(1<<i))
        {
```

```
                p+=1<<(r-i-1);
            }
        }
        FD[j]=X1[p];
    }

    // 释放内存
    delete W;
    delete X1;
    delete X2;
}
```

对于FFT的逆变换IFFT，我们可以直接利用傅立叶变换的性质来用FFT完成。函数IFFT()的代码如下：

```
/*************************************************************************
 *
 * 函数名称:
 *   IFFT()
 *
 * 参数:
 *   complex<double> * FD   - 指向频域值的指针
 *   complex<double> * TD   - 指向时域值的指针
 *   r                      -2的幂数
 *
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速傅立叶反变换。
 *
 ************************************************************************/

VOID WINAPI IFFT(complex<double> * FD, complex<double> * TD, int r)
{
    // 傅立叶变换点数
    LONG    count;

    // 循环变量
    int     i;

    complex<double> *X;

    // 计算傅立叶变换点数
    count = 1 << r;

    // 分配运算所需存储器
    X = new complex<double>[count];

    // 将频域点写入X
    memcpy(X, FD, sizeof(complex<double>) * count);
```

```
    // 求共轭
    for(i = 0; i < count; i++)
    {
        X[i] = complex<double> (X[i].real(), -X[i].imag());
    }

    // 调用快速傅立叶变换
    FFT(X, TD, r);

    // 求时域点的共轭
    for(i = 0; i < count; i++)
    {
        TD[i] = complex<double> (TD[i].real() / count, -TD[i].imag() / count);
    }

    // 释放内存
    delete X;
}
```

有了上面的 FFT()和 IFFT()函数，我们就可以非常方便的进行二维离散数字图像的傅立叶变换。下面是实现该功能的子函数 Fourier()的源代码。

```
/*************************************************************************
 *
 * 函数名称：
 *   Fourier()
 *
 * 参数：
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值：
 *   BOOL               - 成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用来对图像进行傅立叶变换。
 *
 ************************************************************************/

BOOL WINAPI Fourier(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 中间变量
    double  dTemp;

    // 循环变量
    LONG    i;
    LONG    j;
```

```
// 进行傅立叶变换的宽度和高度（2的整数次方）
LONG    w;
LONG    h;

int     wp;
int     hp;

// 图像每行的字节数
LONG    lLineBytes;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 赋初值
w = 1;
h = 1;
wp = 0;
hp = 0;

// 计算进行傅立叶变换的宽度和高度（2的整数次方）
while(w * 2 <= lWidth)
{
    w *= 2;
    wp++;
}

while(h * 2 <= lHeight)
{
    h *= 2;
    hp++;
}

// 分配内存
complex<double> *TD = new complex<double>[w * h];
complex<double> *FD = new complex<double>[w * h];

// 行
for(i = 0; i < h; i++)
{
    // 列
    for(j = 0; j < w; j++)
    {
        // 指向DIB第i行，第j个像素的指针
        lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        // 给时域赋值
        TD[j + w * i] = complex<double>(*(lpSrc), 0);
    }
}
```

```
for(i = 0; i < h; i++)
{
    // 对y方向进行快速傅立叶变换
    FFT(&TD[w * i], &FD[w * i], wp);
}

// 保存变换结果
for(i = 0; i < h; i++)
{
    for(j = 0; j < w; j++)
    {
        TD[i + h * j] = FD[j + w * i];
    }
}

for(i = 0; i < w; i++)
{
    // 对x方向进行快速傅立叶变换
    FFT(&TD[i * h], &FD[i * h], hp);
}

// 行
for(i = 0; i < h; i++)
{
    // 列
    for(j = 0; j < w; j++)
    {
        // 计算频谱
        dTemp = sqrt(FD[j * h + i].real() * FD[j * h + i].real() +
                     FD[j * h + i].imag() * FD[j * h + i].imag()) / 100;

        // 判断是否超过255
        if (dTemp > 255)
        {
            // 对于超过的，直接设置为255
            dTemp = 255;
        }

        // 指向DIB第(i<h/2 ? i+h/2 : i-h/2)行，第(j<w/2 ? j+w/2 : j-w/2)个像素的指针
        // 此处不直接取i和j，是为了将变换后的原点移到中心
        //lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        lpSrc = (unsigned char*)lpDIBBits + lLineBytes *
            (lHeight - 1 - (i<h/2 ? i+h/2 : i-h/2)) + (j<w/2 ? j+w/2 : j-w/2);

        // 更新原图像
        * (lpSrc) = (BYTE)(dTemp);
    }
}

// 删除临时变量
```

```
    delete TD;
    delete FD;

    // 返回
    return TRUE;
}
```

接下来我们添加一个名为正交变换的菜单。如图 5－13 所示。该菜单中有三个子菜单，就是本章中将介绍的三种正交变换。

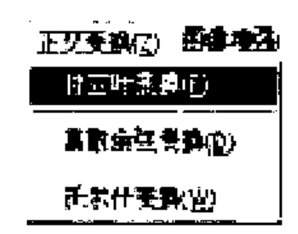


图 5－13 正交变换菜单

在傅立叶变换子菜单的单击处理事件中添加如下代码：

```
void CCh1_1View::OnFreqFour()
{
    // 图像傅立叶变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的傅立叶变换，其它的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {

        // 提示用户
        MessageBox("目前只支持256色位图的傅立叶变换！", "系统提示",
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
```

```
		// 返回
		return;
	}

	// 更改光标形状
	BeginWaitCursor();

	// 调用Fourier()函数进行傅立叶变换
	if (::Fourier(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
	{

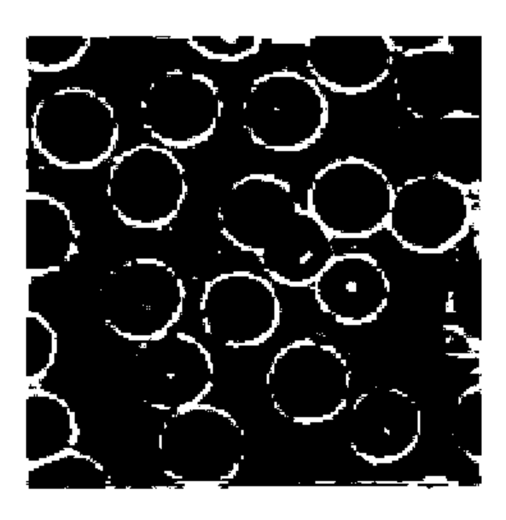
		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);
		// 更新视图
		pDoc->UpdateAllViews(NULL);
	}
	else
	{
		// 提示用户
		MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
	}

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();

}
```

运行上述代码，就可以对一幅二维的数字图像进行傅立叶变换。图 5－14 的左边为变换前的图像，右边是利用上述代码进行变换后的图像。



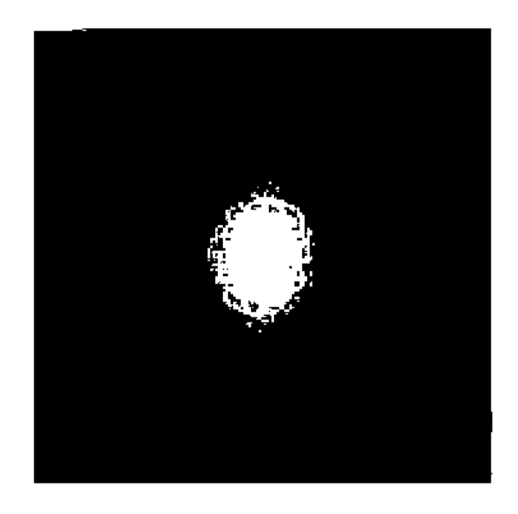


图 5－14　图像的傅立叶变换

5.2 离散余弦变换

尽管傅立叶变换具有很多优点，得到了广泛的应用，但是它也有缺点。例如：傅立叶变换需要计算的是复数而不是实数，一般进行复数运算要比进行实数运算费时得多。如果采用其他合适的完备正交函数系来代替傅立叶变换所利用的正、余弦函数构成的完备正交函数系，就可以避免这种复数运算。本节介绍的离散余弦变换和下节将要介绍的沃尔什变换就是基于实数的正交变换。

5.2.1 离散余弦变换的基本概念

一维离散余弦变换的定义如下：

$$
\begin{aligned}
F(0) &= \frac{1}{\sqrt{N}}\sum_{x=0}^{N-1} f(x) \\
F(\mu) &= \sqrt{\frac{2}{N}}\sum_{x=0}^{N-1} f(x)\cos\frac{(2x+1)\mu\pi}{2N}
\end{aligned}
\tag{5-67}
$$

式中 $F(\mu)$ 为第 μ 个余弦变换系数，μ 为广义频率变量，$\mu=1,2,\cdots,N-1$；$f(x)$ 为时域中 N 点序列，$x=0,1,2,\cdots,N-1$。

一维离散反余弦变换由下式表示：

$$
f(x)=\sqrt{\frac{1}{N}}F(0)+\sqrt{\frac{2}{N}}\sum_{\mu=1}^{N-1}F(\mu)\cos\frac{(2x+1)\mu\pi}{2N}
\tag{5-68}
$$

对于二维的离散余弦变换，它的定义表达式如下：

$$
\begin{aligned}
F(0,0) &= \frac{1}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y) \\
F(\mu,0) &= \frac{\sqrt{2}}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cdot\cos\frac{(2x+1)\mu\pi}{2N} \\
F(0,\nu) &= \frac{\sqrt{2}}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cdot\cos\frac{(2y+1)\nu\pi}{2N} \\
F(\mu,\nu) &= \frac{2}{N}\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cdot\cos\frac{(2x+1)\mu\pi}{2N}\cdot\cos\frac{(2y+1)\nu\pi}{2N}
\end{aligned}
\tag{5-69}
$$

其中 $f(x,y)$ 为空间域中二维向量，$x,y=0,1,2,\cdots,N-1$，$F(\mu,\nu)$ 为变换系数矩阵，$\mu,\nu=1,2,\cdots,N-1$。

二维离散反余弦变换由下式表示：

$$
\begin{aligned}
f(x,y) = & \frac{1}{N}F(0,0) + \frac{\sqrt{2}}{N}\sum_{\mu=1}^{N-1}F(\mu,0)\cdot\cos\frac{(2x+1)\mu\pi}{2N} \\
& + \frac{\sqrt{2}}{N}\sum_{v=1}^{N-1}F(0,v)\cdot\cos\frac{(2y+1)v\pi}{2N} \\
& + \frac{2}{N}\sum_{\mu=1}^{N-1}\sum_{v=1}^{N-1}F(\mu,v)\cdot\cos\frac{(2x+1)\mu\pi}{2N}\cdot\cos\frac{(2y+1)v\pi}{2N}
\end{aligned}
\tag{5-70}
$$

上面是离散余弦变换的解析定义表达式，还有一种更简洁的矩阵定义表达式。下面以 $N=4$ 的一维离散余弦变换为例，按照公式 5—67 可得：

$$
\begin{cases}
F(0) = 0.500f(0)+0.500f(1)+0.500f(2)+0.500f(3) \\
F(1) = 0.653f(0)+0.271f(1)-0.271f(2)-0.653f(3) \\
F(2) = 0.500f(0)-0.500f(1)-0.500f(2)+0.500f(3) \\
F(3) = 0.271f(0)-0.653f(1)+0.653f(2)-0.271f(3)
\end{cases}
\tag{5-71}
$$

写成矩阵表达式为：

$$
\begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \end{bmatrix} =
\begin{bmatrix}
0.500 & 0.500 & 0.500 & 0.500 \\
0.653 & 0.271 & -0.271 & -0.653 \\
0.500 & -0.500 & -0.500 & 0.500 \\
0.271 & -0.653 & 0.653 & -0.271
\end{bmatrix} \cdot
\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{bmatrix}
\tag{5-72}
$$

如果定义 $[A]$ 为变换矩阵，$[F(\mu)]$ 为变换系数向量，$[f(x)]$ 为时域数据向量，则上式可以写成如下形式：

$$
[F(\mu)] = [A]\cdot[f(x)] \tag{5-73}
$$

同理，按照公式 5—68 可得：

$$
\begin{cases}
f(0) = 0.500F(0)+0.653F(1)+0.500F(2)+0.271F(3) \\
f(1) = 0.500F(0)+0.271F(1)-0.500F(2)-0.653F(3) \\
f(2) = 0.500F(0)-0.271F(1)-0.500F(2)+0.653F(3) \\
f(3) = 0.500F(0)-0.653F(1)+0.500F(2)-0.271F(3)
\end{cases}
\tag{5-74}
$$

写成矩阵表达式为：

$$
\begin{bmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \end{bmatrix} =
\begin{bmatrix}
0.500 & 0.653 & 0.500 & 0.271 \\
0.500 & 0.271 & -0.500 & -0.653 \\
0.500 & -0.271 & -0.500 & 0.500 \\
0.500 & -0.653 & 0.500 & -0.271
\end{bmatrix} \cdot
\begin{bmatrix} F(0) \\ F(1) \\ F(2) \\ F(3) \end{bmatrix}
\tag{5-75}
$$

即：

$$[f(x)]=[A]^T\cdot[F(\mu)] \tag{5-76}$$

其中$[A]^T$为矩阵$[A]$的转置。

对于二维离散余弦变换，其矩阵表达式如下：

$$\begin{aligned}&[F(\mu,v)]=[A]\cdot[f(x,y)]\cdot[A]^T\\&[f(x,y)]=[A]^T\cdot[F(\mu,v)]\cdot[A]\end{aligned} \tag{5-77}$$

离散余弦变换来自切比雪夫多项式。切比雪夫多项式的定义如下：

$$\begin{cases}T_0(p)=\sqrt{\dfrac{1}{N}}\\T_\mu(z_x)=\sqrt{\dfrac{2}{N}}\cos[\mu\arccos(z_x)]\end{cases} \tag{5-78}$$

其中$T_\mu(z_x)$是μ和z_x的多项式。如果令$z_x=\cos\dfrac{(2x+1)\pi}{2N}$，那么：

$$T_\mu=\sqrt{\frac{2}{N}}\cos\left[\mu\arccos(\cos\frac{(2x+1)\pi}{2N})\right]=\sqrt{\frac{2}{N}}\cos\frac{(2x+1)\mu\pi}{2N}$$

这和一维离散余弦变换的基向量一致。由于切比雪夫多项式是正交多项式，所以离散余弦变换是一种正交变换。由公式5－72和公式5－75的系数矩阵可以看出，

$$[A]\cdot[A]^T=[I]$$

即满足正交条件。

5.2.2 Visual C++编程实现图像离散余弦变换

要进行离散余弦变换可以从它的定义式出发，但这样做的计算量是相当大的，在实际应用中非常不便，因此需要找一种快速算法。

首先，我们将$f(x)$进行延拓：

$$f_e(x)=\begin{cases}f(x) & x=0,1,2,\cdots,N-1\\0 & x=N,N+1,\cdots,2N-1\end{cases} \tag{5-79}$$

按照一维离散余弦变换的定义，$f_e(x)$的离散余弦变换为：

$$F(0)=\frac{1}{\sqrt{N}}\sum_{x=0}^{2N-1}f_e(x)$$

$$F(\mu)=\sqrt{\frac{2}{N}}\sum_{x=0}^{2N-1}f_e(x)\cos\frac{(2x+1)\mu\pi}{2N}$$
$$=\sqrt{\frac{2}{N}}R_e\left\{\sum_{x=0}^{2N-1}f_e(x)\mathrm{e}^{-\mathrm{j}\frac{(2x+1)\mu\pi}{2N}}\right\} \quad (5-80)$$
$$=\sqrt{\frac{2}{N}}R_e\left\{\mathrm{e}^{-\mathrm{j}\frac{\mu\pi}{2N}}\cdot\sum_{x=0}^{2N-1}f_e(x)\mathrm{e}^{-\mathrm{j}\frac{2x\mu\pi}{2N}}\right\}$$

其中 $R_e\{\ \}$表示取复数的实部。

由于 $\sum_{x=0}^{2N-1}f_e(x)\mathrm{e}^{-\mathrm{j}\frac{2x\mu\pi}{2N}}$ 为 $f_e(x)$ 的 $2N$ 点离散傅立叶变换。因此，在作离散余弦变换时，可以把长度为 N 的序列 $f(x)$ 的长度延拓为 $2N$ 的序列 $f_e(x)$，然后对延拓的结果 $f_e(x)$ 进行离散傅立叶变换，最后去离散傅立叶变换的实部便是离散余弦变换的结果。

同理，对于离散余弦反变换，也可以按照下式延拓 $F(\mu)$：

$$F_e(\mu)=\begin{cases}F(\mu) & \mu=0,1,2,\cdots,N-1\\ 0 & \mu=N,N+1,\cdots,2N-1\end{cases} \quad (5-81)$$

按照公式 5－68 可得：

$$f(x)=\sqrt{\frac{1}{N}}F_e(0)+\sqrt{\frac{2}{N}}\sum_{\mu=1}^{2N-1}F_e(\mu)\cos\frac{(2x+1)\mu\pi}{2N}$$
$$=\sqrt{\frac{1}{N}}F_e(0)+\sqrt{\frac{2}{N}}R_e\left\{\sum_{\mu=1}^{2N-1}F_e(\mu)\mathrm{e}^{-\mathrm{j}\frac{(2x+1)\mu\pi}{2N}}\right\}$$
$$=\sqrt{\frac{1}{N}}F_e(0)+\sqrt{\frac{2}{N}}R_e\left\{\sum_{\mu=1}^{2N-1}F_e(\mu)\mathrm{e}^{-\mathrm{j}\frac{\mu\pi}{2N}}\mathrm{e}^{-\mathrm{j}\frac{2x\mu\pi}{2N}}\right\} \quad (5-82)$$
$$=\left(\sqrt{\frac{1}{N}}-\sqrt{\frac{2}{N}}\right)F_e(0)+\sqrt{\frac{2}{N}}R_e\left\{\sum_{\mu=0}^{2N-1}\left[F_e(\mu)\mathrm{e}^{-\mathrm{j}\frac{\mu\pi}{2N}}\right]\mathrm{e}^{-\mathrm{j}\frac{2x\mu\pi}{2N}}\right\}$$

从公式 5－82 可见，离散余弦反变换可以由 $\left[F_e(\mu)\mathrm{e}^{-\mathrm{j}\frac{\mu\pi}{2N}}\right]$ 的 2N 点的反傅立叶变换来实现。

有了上面的理论基础，我们就可以编写自己的 DCT 和 IDCT 程序。下面定义的函数 DCT() 可以完成一维离散余弦变换。它有三个参数：一个是指向时域数组的指针 f，该数组保存着

要进行离散余弦变换的数值序列，类型为双精度；第二个是指向频域数组的指针，用来保存快速离散余弦变换结果。参数 r 为 $\log_2 N$，即为迭代次数。离散余弦变换的点数可以由参数 r 直接求出，只要将 1 左移 r 位（就是 2^r）即函数 IDCT()和函数 DCT()类似，它的第一个参数是指向频域数组的指针，第二个参数才是指向时域数组的指针。

下面给出函数 DCT()和 IDCT()的完整代码。

```
/*************************************************************************
 *
 * 函数名称:
 *   DCT()
 *
 * 参数:
 *   double * f              - 指向时域数组的指针
 *   double * F              - 指向频域数组的指针
 *   r                       -2的幂数
 *
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速离散余弦变换。该函数利用2N点的快速傅立叶变换
 * 来实现离散余弦变换。
 *
 ************************************************************************/
VOID WINAPI DCT(double *f, double *F, int r)
{
    // 离散余弦变换点数
    LONG    count;

    // 循环变量
    int     i;

    // 中间变量
    double  dTemp;

    complex<double> *X;

    // 计算离散余弦变换点数
    count = 1<<r;

    // 分配内存
    X = new complex<double>[count*2];

    // 赋初值为0
    memset(X, 0, sizeof(complex<double>) * count * 2);

    // 将时域点写入数组X
    for(i=0;i<count;i++)
```

```
    {
        X[i] = complex<double> (f[i], 0);
    }

    // 调用快速傅立叶变换
    FFT(X,X,r+1);

    // 调整系数
    dTemp = 1/sqrt(count);

    // 求F[0]
    F[0] = X[0].real() * dTemp;

    dTemp *= sqrt(2);

    // 求F[u]
    for(i = 1; i < count; i++)
    {
        F[i]=(X[i].real() * cos(i*PI/(count*2)) +
              X[i].imag() * sin(i*PI/(count*2))) * dTemp;
    }

    // 释放内存
    delete X;
}

/*************************************************************************
 *
 * 函数名称:
 *   IDCT()
 *
 * 参数:
 *   double * F              - 指向频域值的指针
 *   double * f              - 指向时域值的指针
 *   r                       -2的幂数
 *
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速离散余弦反变换。该函数也利用2N点的快速傅立叶变换
 * 来实现离散余弦反变换。
 *
 ************************************************************************/
VOID WINAPI IDCT(double *F, double *f, int r)
{
    // 离散余弦反变换点数
    LONG    count;

    // 循环变量
    int     i;
```

```
    // 中间变量
    double  dTemp, d0;

    complex<double> *X;

    // 计算离散余弦变换点数
    count = 1<<r;

    // 分配内存
    X = new complex<double>[count*2];

    // 赋初值为0
    memset(X, 0, sizeof(complex<double>) * count * 2);

    // 将频域变换后点写入数组X
    for(i=0;i<count;i++)
    {
        X[i] = complex<double> (F[i] * cos(i*PI/(count*2)), F[i] * sin(i*PI/(count*2)));
    }

    // 调用快速傅立叶反变换
    IFFT(X,X,r+1);

    // 调整系数
    dTemp = sqrt(2.0/count);
    d0 = (sqrt(1.0/count) - dTemp) * F[0];

    // 计算f(x)
    for(i = 0; i < count; i++)
    {
        f[i] = d0 + X[i].real()* dTemp * 2 * count;
    }

    // 释放内存
    delete X;
}
```

有了上面的 DCT()函数，我们下面就可以方便的进行二维图像的离散余弦变换。下面是实现该功能的子函数 DIBDct ()的源代码。

```
/*************************************************************************
 *
 * 函数名称:
 *   DIBDct()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
```

```
 *   BOOL                    - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行离散余弦变换。
 *
 ************************************************************************/
BOOL WINAPI DIBDct(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 进行傅立叶变换的宽度和高度（2的整数次方）
    LONG    w;
    LONG    h;

    // 中间变量
    double  dTemp;

    int     wp;
    int     hp;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 赋初值
    w = 1;
    h = 1;
    wp = 0;
    hp = 0;

    // 计算进行离散余弦变换的宽度和高度（2的整数次方）
    while(w * 2 <= lWidth)
    {
        w *= 2;
        wp++;
    }

    while(h * 2 <= lHeight)
    {
        h *= 2;
        hp++;
    }
```

```
// 分配内存
double *f = new double[w * h];
double *F = new double[w * h];

// 行
for(i = 0; i < h; i++)
{
    // 列
    for(j = 0; j < w; j++)
    {
        // 指向DIB第i行，第j个像素的指针
        lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        // 给时域赋值
        f[j + i * w] = *(lpSrc);
    }
}

for(i = 0; i < h; i++)
{
    // 对y方向进行离散余弦变换
    DCT(&f[w * i], &F[w * i], wp);
}

// 保存计算结果
for(i = 0; i < h; i++)
{
    for(j = 0; j < w; j++)
    {
        f[j * h + i] = F[j + w * i];
    }
}

for(j = 0; j < w; j++)
{
    // 对x方向进行离散余弦变换
    DCT(&f[j * h], &F[j * h], hp);
}

// 行
for(i = 0; i < h; i++)
{
    // 列
    for(j = 0; j < w; j++)
    {
        // 计算频谱
        dTemp = fabs(F[j*h+i]);

        // 判断是否超过255
        if (dTemp > 255)
        {
```

```
                // 对于超过的，直接设置为255
                dTemp = 255;
            }

            // 指向DIB第y行，第x个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 更新原图像
            * (lpSrc) = (BYTE)(dTemp);
        }
    }

    // 释放内存
    delete f;
    delete F;

    // 返回
    return TRUE;
}
```

下面我们来编写离散余弦变换子菜单的单击事件代码：

```
void CCh1_1View::OnFreqDct()
{
    // 图像离散余弦变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的离散余弦变换，其它的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的离散余弦变换！", "系统提示",
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
```

```
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 调用DIBDct()函数进行离散余弦变换
    if (::DIBDct(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

运行上述代码，就可以对一幅二维的数字图像进行离散余弦变换。图 5－15 就是利用上述代码进行变换前后的图像。

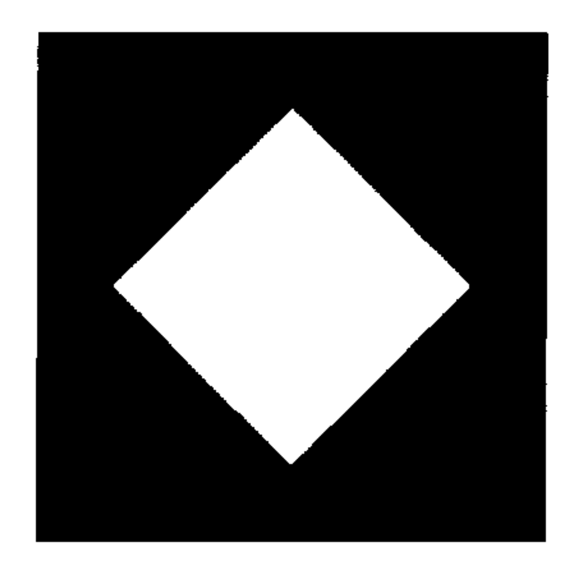



图 5－15 图像的离散余弦变换

5.3 沃尔什变换

离散傅立叶变换和离散余弦变换虽然有快速算法，但它们还是都用到了复数乘法，所以运算速度仍然比较慢。在某些应用领域中，有时需要更有效的变换方法，沃尔什变换就是其中一种。沃尔什变换最主要的优点在于需要的存储空间小，运算速度快，这在实时处理中是非常重要的。

5.3.1 沃尔什函数

沃尔什函数是在 1923 年由美国数学家沃尔什（Walsh）提出。沃尔什函数系是一个完备的正交函数系，其取值只能是 1 和－1。从排列次序上分，沃尔什函数可以有三种定义方法，一种是按照沃尔什排列（即按列率排列）来定义；另外一种是按照佩列排列（即自然排列）来定义；第三种是按照哈达玛排列来定义。这里我们只介绍最后一种情况：按照哈达玛排列来定义的沃尔什函数。

按照哈达玛排列的沃尔什函数是从 2^n 阶哈达玛矩阵而得。2^n 阶哈达玛矩阵每一行的符合变换规律对应于某个沃尔什函数在正交空间内符号变换的规律。即，2^n 阶哈达玛矩阵的每一行就对应于一个离散沃尔什函数。2^n 阶哈达玛矩阵有如下形式：

$$H(0)=[1] \tag{5-83}$$

$$H(1)=\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{5-84}$$

$$H(2)=\begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & 1 & -1\\ 1 & 1 & -1 & -1\\ 1 & -1 & -1 & 1\end{bmatrix} \tag{5-85}$$

$$\vdots$$

一般，

$$H(m)=\begin{bmatrix}H(m-1) & H(m-1)\\ H(m-1) & -H(m-1)\end{bmatrix}=H(m-1)\otimes H(1) \tag{5-86}$$

式中 $\otimes$ 称为克罗内克（Kronecker）积。公式 5－86 称为哈达玛矩阵的递推关系式，利用该公式可以产生任意的 2^n 阶哈达玛矩阵。

按照哈达玛排列的沃尔什函数用 $wal_H(i,t)$ 来表示。它的前 8 个函数的波形如图 5－16 所示。写成矩阵形式如公式 5－87 所示。

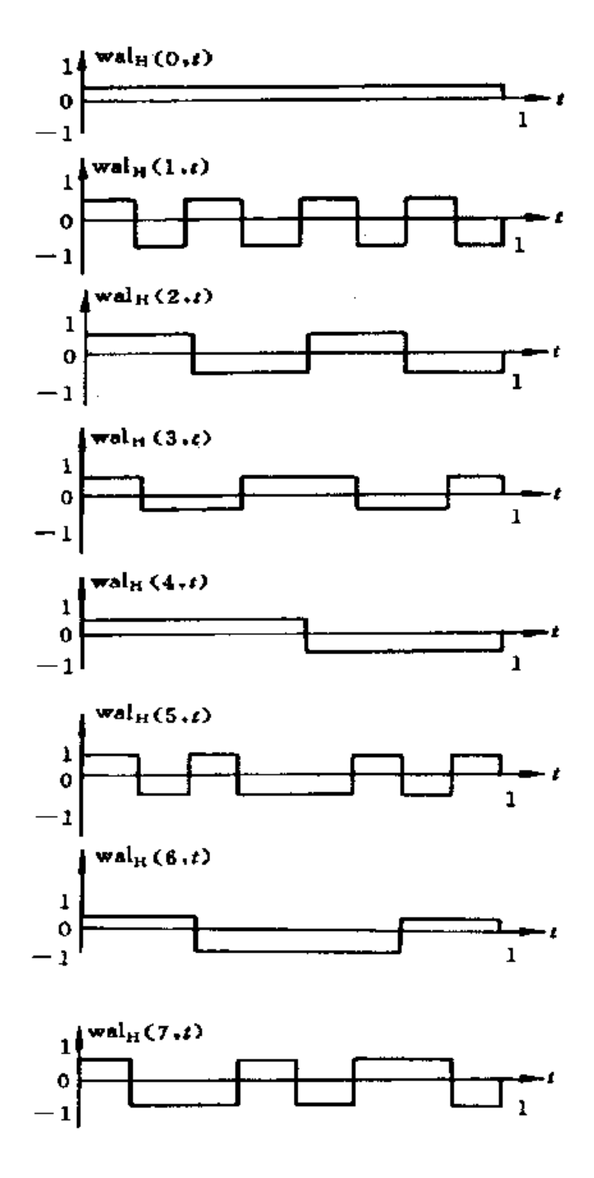


图 5−16 按哈达玛排列的沃尔什函数

$$
H_H(3)=\begin{bmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1\\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1\\
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1\\
1 & -1 & 1 & -1 & 1 & -1 & 1 & -1\\
1 & 1 & -1 & -1 & 1 & 1 & -1 & -1\\
1 & -1 & -1 & 1 & 1 & -1 & -1 & 1
\end{bmatrix} \tag{5-87}
$$

一般，按照哈达玛排列的沃尔什函数有如下规律：

(1) 从2^n阶哈达玛矩阵中可以得2^n个沃尔什函数。

(2) 从不同阶数的哈达玛矩阵得到的沃尔什函数排列顺序是不同的。例如，从$H_H(4)$得到的沃尔什函数$wal_H(2,t)$并不等于从$H_H(8)$中得到的$wal_H(2,t)$，而是从$H_H(8)$中得到的$wal_H(4,t)$。

沃尔什函数有一些非常重要的性质如下：

(1) 在区间 [0, 1] 内下列等式成立：

$$\int_0^1 wal(0,t)\mathrm{d}t = 1 \tag{5-88}$$

$$\int_0^1 wal(i,t)\mathrm{d}t = 0 \tag{5-89}$$

$$|wal(i,t)| = 1 \tag{5-90}$$

该性质说明在区间 [0, 1] 内，除了$wal(0,t)$外，其它沃尔什函数取+1 和-1 的次数是相同的，即积分为 0。

(2) 在区间 [0, 1] 内第一小段时间内（通常称为时隙）沃尔什函数总是取值+1。

(3) 沃尔什函数有如下乘法定理：

$$wal(i,t)\cdot wal(j,t) = wal(i \oplus j,t) \tag{5-91}$$

其中$\oplus$表示模 2 加法。

(4) 沃尔什函数有归一化正交性：

$$\int_0^1 wal(i,t)\cdot wal(j,t)dt = \begin{cases} 0 & i \neq j \\ 1 & i = j \end{cases} \tag{5-92}$$

(5) $wal_H(2^n i,t) = wal_H(i,2^n t)$

(6) 对称性：$wal_H(i,t) = wal_H(t,i)$（只适用于离散沃尔什函数）

5.3.2 沃尔什变换

离散沃尔什变换可以由公式 5－93 来确定：

$$W(i) = \frac{1}{N}\sum_{t=0}^{N-1} f(t)\cdot wal(i,t) \tag{5-93}$$

$$f(t) = \sum_{i=0}^{N-1} W(i)\cdot wal(i,t) \tag{5-94}$$

写成矩阵表达式为：

$$\begin{bmatrix} W(0) \\ W(1) \\ \vdots \\ W(N-1) \end{bmatrix} = \frac{1}{N}\left[wal(N)\right]\begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{bmatrix} \tag{5-95}$$

$$\begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{bmatrix} = [wal(N)] \begin{bmatrix} W(0) \\ W(1) \\ \vdots \\ W(N-1) \end{bmatrix} \tag{5-96}$$

其中$[wal(N)]$表示N阶沃尔什矩阵。

此外，沃尔什函数可以写成如下形式：

$$wal(i,t) = (-1)^{\sum_{k=0}^{p-1} t_{p-1-k}(i_{k+1} \oplus i_k)} \tag{5-97}$$

其中，如果将t表示为二进制形式：$t=(t_{p-1}t_{p-2}\cdots t_k \cdots t_1 t_0)_B$，$t_{p-1+k}$ 为t的二进制形式的第$p-1+k$位。同样，i_k为i的二进制形式的第k位。$\oplus$表示模2加法。

因此，沃尔什变换又可以表示为：

$$W(i) = \frac{1}{N}\sum_{t=0}^{N-1} f(t)\cdot(-1)^{\sum_{k=0}^{p-1} t_{p-1-k}(i_{k+1} \oplus i_k)} \tag{5-98}$$

$$f(t) = \sum_{i=0}^{N-1} W(i)\cdot(-1)^{\sum_{k=0}^{p-1} t_{p-1-k}(i_{k+1} \oplus i_k)} \tag{5-99}$$

5.3.3 离散沃尔什一哈达玛变换

由于哈达玛矩阵具有简单的递推关系，它的高阶矩阵可以用低阶矩阵的克罗内克积$\otimes$来获得，这就使按照哈达玛排列的沃尔什变换非常简便。因此，应用最多的沃尔什变换就是沃尔什一哈达玛变换。

离散沃尔什一哈达玛变换的定义可以由公式5一95和公式5一96直接得到：

$$\begin{bmatrix} W(0) \\ W(1) \\ \vdots \\ W(N-1) \end{bmatrix} = \frac{1}{N}[H(N)] \begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{bmatrix} \tag{5-100}$$

$$\begin{bmatrix} f(0) \\ f(1) \\ \vdots \\ f(N-1) \end{bmatrix} = [H(N)] \begin{bmatrix} W(0) \\ W(1) \\ \vdots \\ W(N-1) \end{bmatrix} \tag{5-101}$$

其中$[H(N)]$表示N阶哈达玛矩阵。

5.3.4 快速沃尔什一哈达玛变换

和离散傅立叶变换相似，沃尔什变换也有其快速算法。利用沃尔什变换快速算法，完成一次变换只需要$N\log_2 N$次加减法，运算速度大大提高。下面介绍一下离散沃尔什一哈达

玛变换的快速算法。

下面以 8 阶离散沃尔什—哈达玛变换为例，由哈达玛矩阵的性质可知：

$$
\begin{aligned}
[H_8]&=[H_2]\otimes[H_4]\\
&=\begin{bmatrix} H_4 & H_4 \\ H_4 & -H_4 \end{bmatrix}\\
&=\begin{bmatrix} H_4 & 0 \\ 0 & -H_4 \end{bmatrix}\begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}\\
&=\begin{bmatrix} H_2 & H_2 & 0 & 0 \\ H_2 & -H_2 & 0 & 0 \\ 0 & 0 & H_2 & H_2 \\ 0 & 0 & H_2 & -H_2 \end{bmatrix}\begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}\\
&=\begin{bmatrix} H_2 & 0 & 0 & 0 \\ 0 & H_2 & 0 & 0 \\ 0 & 0 & H_2 & 0 \\ 0 & 0 & 0 & H_2 \end{bmatrix}\begin{bmatrix} I_2 & I_2 & 0 & 0 \\ I_2 & -I_2 & 0 & 0 \\ 0 & 0 & I_2 & I_2 \\ 0 & 0 & I_2 & -I_2 \end{bmatrix}\begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix}
\end{aligned}
\tag{5-102}
$$

如果令：

$$
[G_0]=\begin{bmatrix} H_2 & 0 & 0 & 0 \\ 0 & H_2 & 0 & 0 \\ 0 & 0 & H_2 & 0 \\ 0 & 0 & 0 & H_2 \end{bmatrix} \tag{5-103}
$$

$$
[G_1]=\begin{bmatrix} I_2 & I_2 & 0 & 0 \\ I_2 & -I_2 & 0 & 0 \\ 0 & 0 & I_2 & I_2 \\ 0 & 0 & I_2 & -I_2 \end{bmatrix} \tag{5-104}
$$

$$
[G_2]=\begin{bmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{bmatrix} \tag{5-105}
$$

则：$[H_8]=[G_0][G_1][G_2]$

其中 I_2 和 I_4 为么阵（单位矩阵）。

如果再令：

$$
[f_1(t)]=[G_2][f(t)] \tag{5-106}
$$

$$[f_2(t)]=[G_1][f_1(t)] \tag{5-107}$$

$$[f_3(t)]=[G_0][f_2(t)] \tag{5-108}$$

那么，8 阶离散沃尔什一哈达玛变换公式为：

$$[W_H(n)]=\frac{1}{8}[G_0][G_1][G_2][f(t)]=\frac{1}{8}[f_3(t)] \tag{5-109}$$

下面是计算$[f_1(t)]$，$[f_2(t)]$和$[f_3(t)]$的具体公式：

$$\begin{bmatrix} f_1(0)\\ f_1(1)\\ f_1(2)\\ f_1(3)\\ f_1(4)\\ f_1(5)\\ f_1(6)\\ f_1(7) \end{bmatrix}=\begin{bmatrix} 1&0&0&0&1&0&0&0\\ 0&1&0&0&0&1&0&0\\ 0&0&1&0&0&0&1&0\\ 0&0&0&1&0&0&0&1\\ 1&0&0&0&-1&0&0&0\\ 0&1&0&0&0&-1&0&0\\ 0&0&1&0&0&0&-1&0\\ 0&0&0&1&0&0&0&-1 \end{bmatrix}\begin{bmatrix} f(0)\\ f(1)\\ f(2)\\ f(3)\\ f(4)\\ f(5)\\ f(6)\\ f(7) \end{bmatrix}=\begin{bmatrix} f(0)+f(4)\\ f(1)+f(5)\\ f(3)+f(6)\\ f(3)+f(7)\\ f(0)-f(4)\\ f(1)-f(5)\\ f(2)-f(6)\\ f(3)-f(7) \end{bmatrix} \tag{5-110}$$

$$\begin{bmatrix} f_2(0)\\ f_2(1)\\ f_2(2)\\ f_2(3)\\ f_2(4)\\ f_2(5)\\ f_2(6)\\ f_2(7) \end{bmatrix}=\begin{bmatrix} 1&0&1&0&0&0&0&0\\ 0&1&0&1&0&0&0&0\\ 1&0&-1&0&0&0&0&0\\ 0&1&0&-1&0&0&0&0\\ 0&0&0&0&1&0&1&0\\ 0&0&0&0&0&1&0&1\\ 0&0&0&0&1&0&-1&0\\ 0&0&0&0&0&1&0&-1 \end{bmatrix}\begin{bmatrix} f_1(0)\\ f_1(1)\\ f_1(2)\\ f_1(3)\\ f_1(4)\\ f_1(5)\\ f_1(6)\\ f_1(7) \end{bmatrix}=\begin{bmatrix} f_1(0)+f_1(2)\\ f_1(1)+f_1(3)\\ f_1(0)-f_1(2)\\ f_1(1)-f_1(3)\\ f_1(4)+f_1(6)\\ f_1(5)+f_1(7)\\ f_1(4)-f_1(6)\\ f_1(5)-f_1(7) \end{bmatrix} \tag{5-111}$$

$$\begin{bmatrix} f_3(0)\\ f_3(1)\\ f_3(2)\\ f_3(3)\\ f_3(4)\\ f_3(5)\\ f_3(6)\\ f_3(7) \end{bmatrix}=\begin{bmatrix} 1&1&0&0&0&0&0&0\\ 1&-1&0&0&0&0&0&0\\ 0&0&1&1&0&0&0&0\\ 0&0&1&-1&0&0&0&0\\ 0&0&0&0&1&1&0&0\\ 0&0&0&0&1&-1&0&0\\ 0&0&0&0&0&0&1&1\\ 0&0&0&0&0&0&1&-1 \end{bmatrix}\begin{bmatrix} f_2(0)\\ f_2(1)\\ f_2(2)\\ f_2(3)\\ f_2(4)\\ f_2(5)\\ f_2(6)\\ f_2(7) \end{bmatrix}=\begin{bmatrix} f_2(0)+f_2(1)\\ f_2(0)-f_2(1)\\ f_2(2)+f_2(3)\\ f_2(2)-f_2(3)\\ f_2(4)+f_2(5)\\ f_2(4)-f_2(5)\\ f_2(6)+f_2(7)\\ f_2(6)-f_2(7) \end{bmatrix} \tag{5-112}$$

这样就可以用蝶形流图来进行计算。如图 5－17 所示。

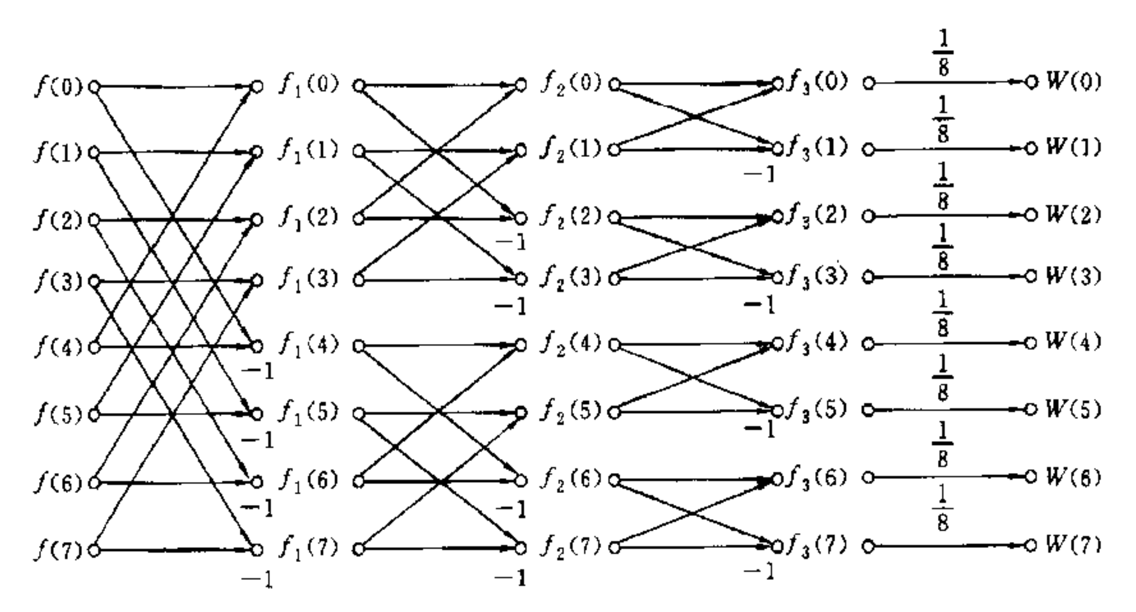


图 5－17　快速沃尔什－哈达玛变换蝶形流图一

和快速傅立叶变换类似，快速沃尔什－哈达玛变换也还有另外一种蝶形流图。推导如下：由于$[H_8]$, $[G_0]$, $[G_1]$和$[G_2]$都是对称矩阵即：

$$[H_8]=[H_8]^{\mathrm{T}} \qquad [G_1]=[G_1]^{\mathrm{T}}$$

$$[G_0]=[G_0]^{\mathrm{T}} \qquad [G_2]=[G_2]^{\mathrm{T}}$$

所以：

$$[H_8]=[H_8]^T=\{[G_0][G_1][G_2]\}^T=[G_2]^T[G_1]^T[G_0]^T=[G_2][G_1][G_0]$$

如果令：

$$[f_1'(t)]=[G_0][f(t)]$$

$$[f_2'(t)]=[G_1][f_1'(t)]$$

$$[f_3'(t)]=[G_2][f_2'(t)]$$

那么，8 阶离散沃尔什－哈达玛变换公式为：

$$[W_H(n)]=\frac{1}{8}[G_2][G_1][G_0][f(t)]=\frac{1}{8}[f_3'(t)]$$

这样就可以得到另外一种蝶形流图。如图 5－18 所示。

对于一般情况，$N=2^p$（p 为正整数），则矩阵$[H_{2^p}]$可以分解成 p 个矩阵$[G_p]$之乘积。即：

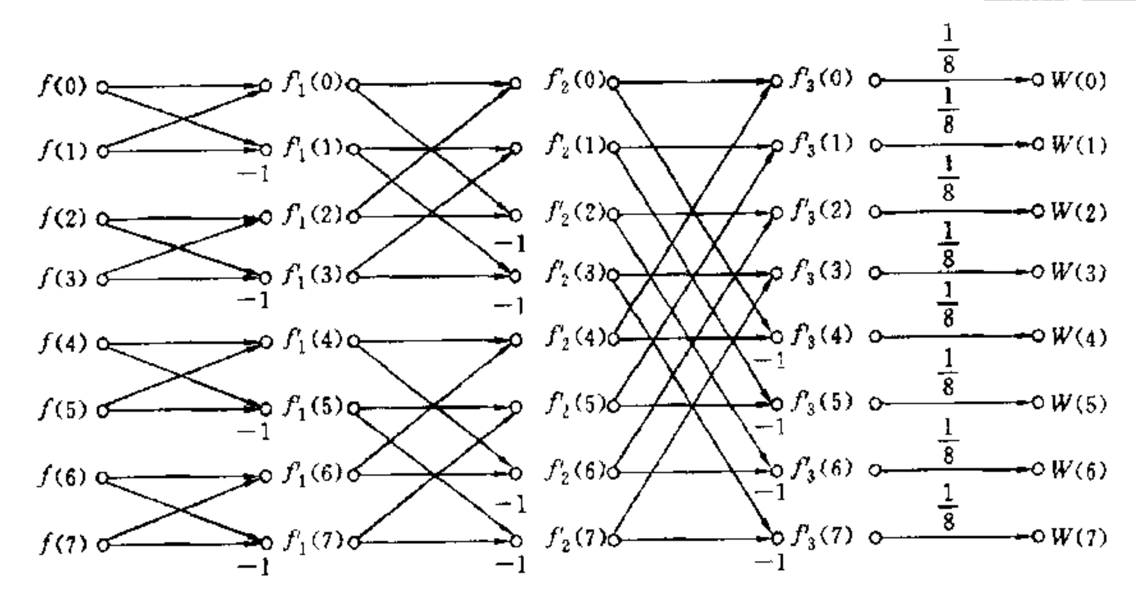


图 5—18 快速沃尔什—哈达玛变换蝶形流图

$$[H_{2^p}]=\prod_{i=0}^{p-1}[G_i]=[G_0][G_1][G_2]\cdots[G_{p-1}]$$
$$=[G_{p-1}][G_{p-2}]\cdots[G_1][G_0]$$

对于2^p阶快速沃尔什—哈达玛变换的蝶形流图也可用上述方法引伸。

在数字图像处理中，要进行变换的是二维的图像，因此需要引入二维沃尔什—哈达玛变换：

$$W_{xy}(\mu,\nu)=\frac{1}{N_x}\frac{1}{N_y}\sum_{y=0}^{N_y-1}\sum_{x=0}^{N_x-1}f(x,y)\cdot(-1)^{\sum_{r=0}^{p_x-1}x_r\mu_r+\sum_{s=0}^{p_y-1}y_s\nu_s} \tag{5—113}$$

$$f(x,y)=\sum_{\nu=0}^{N_y-1}\sum_{\mu=0}^{N_x-1}W_{xy}(\mu,\nu)\cdot(-1)^{\sum_{r=0}^{p_x-1}x_r\mu_r+\sum_{s=0}^{p_y-1}y_s\nu_s} \tag{5—114}$$

其中，$f(x,y)$代表图像的像素，x,y为该像素在空间中的位置坐标，$W_{xy}(\mu,\nu)$表示变换系数，$N_x=2^{P_x}$，$N_y=2^{P_y}$（P_x,P_y为正整数），$x,\mu=0,1,2,\cdots,N_x-1$，x_r,μ_r为x,μ的二进制码的第r位数字；$y,\nu=0,1,2,\cdots,N_y-1$，y_s,ν_s为y,ν的二进制码的第s位数字。

将$f(x,y)$写成矩阵形式：

$$f(x,y)=\begin{bmatrix} f(0,0) & f(0,1) & \cdots & f(0,N_y-1) \\ f(1,0) & f(1,1) & \cdots & f(1,N_y-1) \\ \cdots & \cdots & \cdots & \cdots \\ f(N_x-1,0) & f(N_x-1,1) & \cdots & f(N_x-1,N_y-1) \end{bmatrix} \tag{5-115}$$

如果令：

$$W_x(\mu\mu,y=\frac{1}{N_x}\sum_{x=0}^{N_x-1}f(x,y)\cdot(-1)^{\sum_{r=0}^{p_x-1}x_r\mu_r} \tag{5-116}$$

那么：

$$W_{xy}(\mu\mu,v=\frac{1}{N_y}\sum_{y=0}^{N_y-1}W_x(\mu\mu,y\cdot(-1)^{\sum_{s=0}^{p_y-1}y_s v_s} \tag{5-117}$$

由上面两个公式可以看出，二维沃尔什－哈达玛变换可以用二次一维沃尔什哈达玛变换来计算。步骤如下：

(1) 以$N=N_x$，对$[f(x,y)]$中N_y个列中的每一列进行一维沃尔什－哈达玛变换，得到$[W_x(\mu,y)]$；

(2) 以$N=N_y$，对$[W_x(\mu,y)]$中N_x个行中的每一行进行一维沃尔什－哈达玛变换，得到$[W_{xy}(\mu,v)]$。

此外，还有另外一种计算方法，是将二维沃尔什－哈达玛变换当作一维来计算：将数据矩阵的各列元素依次顺序排列，这样就形成了$N_x\cdot N_y$个元素的列向量，然后再对该向量进行一维沃尔什－哈达玛变换，最后将变换结果还原成$N_x\times N_y$的矩阵。这两种计算方法所得到的结果是完全相同的。

5.3.5 Visual C++编程实现图像沃尔什－哈达玛变换

有了上面的理论基础，就可以来编写沃尔什－哈达玛变换和反变换程序了。下面定义的函数 WALSH ()可以完成一维沃尔什变换。它有三个参数：一个是指向时域数组的指针 f，该数组保存着要进行沃尔什变换的数值序列，类型为双精度；第二个是指向频域数组的指针，用来保存变换结果。参数 r 为$\log_2 N$，即为迭代次数。变换的点数可以由参数 r 直接求出，只要将 1 左移 r 位（就是2^r）即可。函数 IWALSH ()是进行沃尔什－哈达玛反变换。它直接调用了 WALSH ()函数来实现反变换。

下面给出函数 WALSH ()和 IWALSH ()的完整代码。

```
/*************************************************************************
 *
 * 函数名称：
 *   WALSH()
 *
 * 参数:
 *   double * f            - 指向时域值的指针
 *   double * F            - 指向频域值的指针
 *   r                     －2的幂数
 *
```

```
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速沃尔什-哈达玛变换。
 *
 ************************************************************************/

VOID WINAPI WALSH(double *f, double *F, int r)
{
    // 沃尔什-哈达玛变换点数
    LONG    count;

    // 循环变量
    int     i,j,k;

    // 中间变量
    int     bfsize,p;

    double *X1,*X2,*X;

    // 计算快速沃尔什变换点数
    count = 1 << r;

    // 分配运算所需的数组
    X1 = new double[count];
    X2 = new double[count];

    // 将时域点写入数组X1
    memcpy(X1, f, sizeof(double) * count);

    // 蝶形运算
    for(k = 0; k < r; k++)
    {
        for(j = 0; j < 1<<k; j++)
        {
            bfsize = 1 << (r-k);
            for(i = 0; i < bfsize / 2; i++)
            {
                p = j * bfsize;
                X2[i + p] = X1[i + p] + X1[i + p + bfsize / 2];
                X2[i + p + bfsize / 2] = X1[i + p] - X1[i + p + bfsize / 2];
            }
        }

        // 互换X1和X2
        X = X1;
        X1 = X2;
        X2 = X;
    }
```

```
    // 调整系数
    for(j = 0; j < count; j++)
    {
        p = 0;
        for(i = 0; i < r; i++)
        {
            if (j & (1<<i))
            {
                p += 1 << (r-i-1);
            }
        }

        F[j] = X1[p] / count;
    }

    // 释放内存
    delete X1;
    delete X2;
}

/*************************************************************************
 *
 * 函数名称:
 *   IWALSH()
 *
 * 参数:
 *   double * F              - 指向频域值的指针
 *   double * f              - 指向时域值的指针
 *   r                       -2的幂数
 *
 * 返回值:
 *   无。
 *
 * 说明:
 *   该函数用来实现快速沃尔什-哈达玛反变换。
 *
 ************************************************************************/
VOID WINAPI IWALSH(double *F, double *f, int r)
{
    // 变换点数
    LONG    count;

    // 循环变量
    int     i;

    // 计算变换点数
    count = 1 << r;

    // 调用快速沃尔什-哈达玛变换进行反变换
    WALSH(F, f, r);
```

```
    // 调整系数
    for(i = 0; i < count; i++)
    {
        f[i] *= count;
    }
}
```

有了上面的WALSH()函数，我们下面就可以方便的进行二维图像的沃尔什变换了，下面是实现该功能的子函数DIBWalsh ()和DIBWalsh 1()的源代码。两者的功能是一样的，前者采用了对图像每列每行进行一维沃尔什－哈达玛变换，而后者是将二维图像转换成一个列向量，然后再进行一次一维沃尔什－哈达玛变换。

```
/*************************************************************************
*
 * 函数名称:
 *   DIBWalsh()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行沃尔什-哈达玛变换。函数首先对图像每列进行一维
 * 沃尔什－哈达玛变换，然后对变换结果的每行进行一维沃尔什－哈达玛变换。
 *
 ************************************************************************/

BOOL WINAPI DIBWalsh(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 进行傅立叶变换的宽度和高度（2的整数次方）
    LONG    w;
    LONG    h;

    // 中间变量
    double  dTemp;

    int     wp;
    int     hp;
```

```
// 图像每行的字节数
LONG    lLineBytes;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 赋初值
w = 1;
h = 1;
wp = 0;
hp = 0;

// 计算进行离散余弦变换的宽度和高度（2的整数次方）
while(w * 2 <= lWidth)
{
    w *= 2;
    wp++;
}

while(h * 2 <= lHeight)
{
    h *= 2;
    hp++;
}

// 分配内存
double *f = new double[w * h];
double *F = new double[w * h];

// 行
for(i = 0; i < h; i++)
{
    // 列
    for(j = 0; j < w; j++)
    {
        // 指向DIB第i行，第j个像素的指针
        lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        // 给时域赋值
        f[j + i * w] = *(lpSrc);
    }
}

for(i = 0; i < h; i++)
{
    // 对y方向进行沃尔什-哈达玛变换
    WALSH(f + w * i, F + w * i, wp);
}

// 保存计算结果
for(i = 0; i < h; i++)
```

```
	{
		for(j = 0; j < w; j++)
		{
			f[j * h + i] = F[j + w * i];
		}
	}

	for(j = 0; j < w; j++)
	{
		// 对x方向进行沃尔什-哈达玛变换
		WALSH(f + j * h, F + j * h, hp);
	}

	// 行
	for(i = 0; i < h; i++)
	{
		// 列
		for(j = 0; j < w; j++)
		{
			// 计算频谱
			dTemp = fabs(F[j * h + i] * 1000);

			// 判断是否超过255
			if (dTemp > 255)
			{
				// 对于超过的，直接设置为255
				dTemp = 255;
			}

			// 指向DIB第i行，第j个像素的指针
			lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

			// 更新原图像
			* (lpSrc) = (BYTE)(dTemp);
		}
	}

	//释放内存
	delete f;
	delete F;

	// 返回
	return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   DIBWalsh1()
 *
 * 参数:
```

```
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LONG  lWidth        - 原图像宽度（像素数）
 *   LONG  lHeight       - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL                - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行沃尔什-哈达玛变换。与上面不同的是，此处是将二维
 * 矩阵转换成一个列向量，然后对该列向量进行一次一维沃尔什-哈达玛变换。
 *
 ************************************************************************/

BOOL WINAPI DIBWalsh1(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 进行傅立叶变换的宽度和高度（2的整数次方）
    LONG    w;
    LONG    h;

    // 中间变量
    double  dTemp;

    int     wp;
    int     hp;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 赋初值
    w = 1;
    h = 1;
    wp = 0;
    hp = 0;

    // 计算进行离散余弦变换的宽度和高度（2的整数次方）
    while(w * 2 <= lWidth)
    {
        w *= 2;
        wp++;
    }
```

```
while(h * 2 <= lHeight)
{
    h *= 2;
    hp++;
}

// 分配内存
double *f = new double[w * h];
double *F = new double[w * h];

// 列
for(i = 0; i < w; i++)
{
    // 行
    for(j = 0; j < h; j++)
    {
        // 指向DIB第j行，第i个像素的指针
        lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - j) + i;

        // 给时域赋值
        f[j + i * w] = *(lpSrc);
    }
}

// 调用快速沃尔什—哈达玛变换
WALSH(f, F, wp + hp);

// 列
for(i = 0; i < w; i++)
{
    // 行
    for(j = 0; j < h; j++)
    {
        // 计算频谱
        dTemp = fabs(F[i * w + j] * 1000);

        // 判断是否超过255
        if (dTemp > 255)
        {
            // 对于超过的，直接设置为255
            dTemp = 255;
        }

        // 指向DIB第j行，第i个像素的指针
        lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - j) + i;

        // 更新原图像
        * (lpSrc) = (BYTE)(dTemp);
    }
}
```

```
    //释放内存
    delete f;
    delete F;

    // 返回
    return TRUE;
}
```

下面我们来编写沃尔什变换子菜单的单击事件代码：

```
void CCh1_1View::OnFreqWalh()
{
    // 图像沃尔什-哈达玛变换

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的沃尔什-哈达玛变换，其它类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的沃尔什-哈达玛变换！", "系统提示" ,
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 调用DIBWalsh()或者DIBWalsh1()函数进行变换
    if (::DIBWalsh1(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
```

```
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

运行上述代码，就可以对一幅二维的数字图像进行沃尔什变换。图 5—19 就是利用上述代码进行变换前后的图像。

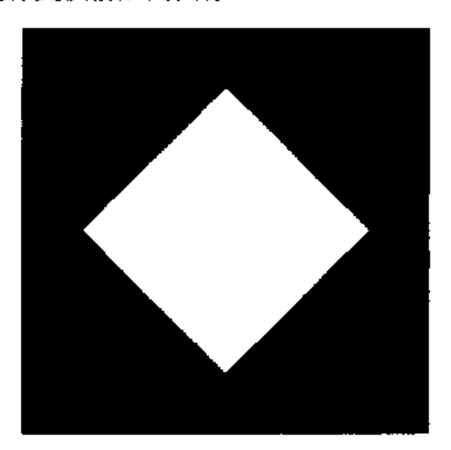




图 5—19 图像的沃尔什变换

第六章　图像的增强

一般情况下，各类图像系统中图像的传送和转换（如成像、复制、扫描、传输以及显示等）总要造成图像的某些降质。例如：在摄像时，光学系统的失真、相对运动、大气流动等都会使图像模糊；在传输过程中，由于噪声污染，图像质量会有所下降。必须对这些降质的图像进行改善处理。通常改善方法有两类：一类是不考虑图像降质的原因，只将图像中感兴趣的特征有选择的突出，而衰减其次要信息；另一类是针对图像降质的原因，设法去补偿降质因素，从而使改善后的图像尽可能的逼近原始图像。第一类方法能提高图像的可读性，改善后的图像不一定逼近原始图像，如突出目标的轮廓，衰减各种噪声，将黑白图像转换成彩色图像等。这类方法通常称为图像增强技术。第二类方法能提高图像质量的逼真度，一般称为图像复原技术。本章中将要重点介绍一些常用的图像增强技术，图像的复原技术将在后面的章节中介绍。

图像的增强技术通常有两类方法：空间域法和频率域法。空间域法主要是在空间域中对图像像素灰度值直接进行运算处理。例如：将包含某点的一个小区域内的各点灰度值进行平均计算，用所得的平均值来代替该点的灰度值，这就是通常所说的平滑处理。空间域法的图像增强技术可以用下式来描述：

$$g(x, y) = f(x, y) \cdot h(x, y)$$

其中 $f(x, y)$ 是处理前的图像；$g(x, y)$ 表示处理后的图像；$h(x, y)$ 为空间运算函数。

图像增强的频率域法就是在图像的某种变换域中（通常是频率域中）对图像的变换值进行某种运算处理，然后变换回空间域。例如：可以先对图像进行傅立叶变换，再对图像的频谱进行某种修正（如滤波等），最后再将修正后的图像进行傅立叶反变换回空间域中，从而增强该图像。它是一种间接处理方法，我们可以用图 6－1 来描述该过程。

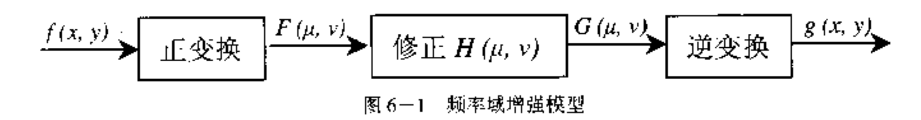


图 6－1　频率域增强模型

其数学描述如下：

$$F(\mu,v) = \Re\{f(x, y)\}$$

$$G(\mu,v) = H(\mu,v) \cdot F(\mu,v)$$

$$g(x, y) = \Re^{-1}\{G(\mu,v)\}$$

其中 $\Re\{\ \}$ 表示某种频域正变换，$\Re^{-1}\{\ \}$ 表示该频域变换的反变换。$F(\mu, v)$ 为原始图像 $f(x, y)$ 结果频域正变换的结果，$H(\mu, v)$ 为频域中的修正函数，$G(\mu, v)$ 是修正后的结果，$g(x, y)$ 是 $G(\mu, v)$ 反变换的结果，即增强后的图像。

6.1 图像的灰度修正

灰度修正使图像在空间域中增强的简单而有效的方法。通常根据图像不同的降质现象而采用不同的修正方法。常见的方法主要有以下 3 种：

(1) 针对图像成像不均匀（如图像半边暗半边亮）而对图像逐点进行不同程度的灰度级校正，目的是使图像灰度均匀。

(2) 针对图像某部分或者整体曝光不足而进行灰度级校正，目的是增加图像的灰度对比度。

(3) 最后一种方法就是直方图修正，它能使图像具有期望的灰度分布，从而有选择的突出所需要的图像特征。

在第三章中介绍图像的点运算时已经描述了一些常用的图像灰度修正方法，在这里就不再重复描述。

6.2 图像的平滑

图像的平滑是一种实用的数字图像处理技术，主要目的是为了减少图像的噪声。一般情况下，在空间域内可以用领域平均来减少噪声；在频率域，由于噪声频谱通常多在高频段，因此可以采用各种形式的低通滤波的办法来减少噪声。

在介绍图像平滑之前，首先介绍一下模板操作。

6.2.1 模板操作

模板操作是数字图像处理中经常用到的一种运算方式，图像的平滑、锐化以及后面将要介绍的细化、边缘检测都要用到模板操作。例如：有一种常见的平滑算法是将原图中一个像素的灰度值和它周围邻近八个像素的灰度值相加，然后将求得的平均值（除以 9）作为新图中该像素的灰度值。我们用如下方法来表示该操作：

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1\bullet & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

上式有点类似于矩阵，我们通常称之为模板(Template)。中间的黑点表示该元素为中心元素，即该个元素是要进行处理的元素。如果模板是：

$$\frac{1}{9}\begin{bmatrix} 1\bullet & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

则该操作应该描述为：将原图中一个像素的灰度值和它右下邻近的 8 个像素的灰度值相加，然后将求得的平均值（除以 9）作为新图中该像素的灰度值。

如果模板为$\begin{bmatrix}2\bullet\\1\end{bmatrix}$，则表示将自身灰度值的 2 倍加下边的元素灰度值作为新值，而$\begin{bmatrix}2\\1\bullet\end{bmatrix}$则表示将自身灰度值加上边元素灰度值的 2 倍作为新灰度值。

通常模板不允许移出边界，所以处理后的新图像会比原图小。例如：当模板是$\begin{bmatrix}1\bullet & 0\\0 & 2\end{bmatrix}$，原图灰度值矩阵是$\begin{bmatrix}1&1&1&1&1\\2&2&2&2&2\\3&3&3&3&3\\4&4&4&4&4\end{bmatrix}$时，经过模板操作后的图像为$\begin{bmatrix}5&5&5&5&-\\8&8&8&8&-\\11&11&11&11&-\\-&-&-&-&-\end{bmatrix}$，“－”表示边界上无法进行模板操作的点，一般的做法是复制原图的灰度值，不再进行任何其他的处理。

模板操作实现了一种邻域运算（Neighborhood Operation），即某个像素点的结果不仅和本像素灰度有关，而且和其邻域点的值有关。模板运算在数学中的描述是卷积（或互相关）运算，在这里就不再介绍了，有兴趣的读者可以自行查看相应的数学教材。

模板运算在图像处理中经常要用到，但是当图像很大时，运算量是非常可观的，也非常耗时。以模板$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4\bullet&2\\1&2&1\end{bmatrix}$运算为例，每个像素完成一次模板操作要用 9 次乘法，8 次加法和 1 次除法。对于一幅 $N\times N$（宽度×高度）的图像，就是 $9(N-2)^2$ 次乘法，$8(N-2)^2$ 次加法和 $(N-2)^2$ 次除法操作，算法复杂度为 $O(N^2)$，这对于大图像来说，是非常可怕的。所以，常用的模板并不大，如 3*3，4*4。有很多专用的图像处理系统，用硬件来完成模板运算，大大提高了速度。

另外，可以设法将 2 维模板运算转换成 1 维模板运算，这对速度的提高也是有益的。例如：(2)式可以分解成一个水平模板和一个竖直模板。即：

$$\frac{1}{16}\begin{bmatrix}1&2&1\\2&4\bullet&2\\1&2&1\end{bmatrix}=\frac{1}{4}\begin{bmatrix}1\\2\bullet\\1\end{bmatrix}\times\frac{1}{4}\begin{bmatrix}1&2\bullet&1\end{bmatrix}=\frac{1}{16}\begin{bmatrix}1\\2\bullet\\1\end{bmatrix}\times\begin{bmatrix}1&2\bullet&1\end{bmatrix}$$

这样，改进后将要进行 $6(N-2)(N-1)$次乘法，$4(N-2)(N-1)$次加法，$(N-2)^2$ 次除法操作，减少了不少次乘法和加法运算。

下面我们来举例验证一下该分解算法的可行性，设图像为$\begin{bmatrix}1&2&4&2\\3&2&5&3\\4&6&4&5\\5&6&8&6\end{bmatrix}$，直接经过模板

$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4\bullet & 2\\ 1 & 2 & 1\end{bmatrix}$处理后变为 $\frac{1}{16}\begin{bmatrix}- & - & - & -\\ - & 53 & 61 & -\\ - & 77 & 81 & -\\ - & - & - & -\end{bmatrix}$。但是如果采用分解后的模板来处理，结果为：

$$\frac{1}{16}\begin{bmatrix}1\\ 2\bullet\\ 1\end{bmatrix}\times\begin{bmatrix}1 & 2\bullet & 1\end{bmatrix}\times\begin{bmatrix}1 & 2 & 4 & 2\\ 3 & 2 & 5 & 3\\ 4 & 6 & 4 & 5\\ 5 & 6 & 8 & 6\end{bmatrix}=\frac{1}{16}\begin{bmatrix}1\\ 2\bullet\\ 1\end{bmatrix}\times\begin{bmatrix}- & 9 & 12 & -\\ - & 12 & 15 & -\\ - & 20 & 19 & -\\ - & 25 & 38 & -\end{bmatrix}=\frac{1}{16}\begin{bmatrix}- & - & - & -\\ - & 53 & 61 & -\\ - & 77 & 81 & -\\ - & - & - & -\end{bmatrix}$$

可以发现两种计算方法得出的结果是完全相同的。

下面我们来编写一个通用的图像模板操作函数。首先分析一下该函数需要的参数，显然我们必须得到要处理图像的信息，即指向图像像素的指针以及图像的高宽信息；其次必须指定要进行变换模板的信息，包括模板大小信息、模板系数信息、模板元素数组、模板中心元素的位置信息。该函数的源代码如下所示。

```
/*************************************************************************
 *
 * 函数名称：
 *   Template()
 *
 * 参数:
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LONG  lWidth        - 原图像宽度（像素数）
 *   LONG  lHeight       - 原图像高度（像素数）
 *   int   iTempH        - 模板的高度
 *   int   iTempW        - 模板的宽度
 *   int   iTempMX       - 模板的中心元素X坐标 ( < iTempW - 1)
 *   int   iTempMY       - 模板的中心元素Y坐标 ( < iTempH - 1)
 *   FLOAT * fpArray     - 指向模板数组的指针
 *   FLOAT fCoef         - 模板系数
 *
 * 返回值:
 *   BOOL                - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用指定的模板（任意大小）来对图像进行操作，参数iTempH指定模板
 * 的高度，参数iTempW指定模板的宽度，参数iTempMX和iTempMY指定模板的中心
 * 元素坐标，参数fpArray指定模板元素，fCoef指定系数。
 *
 ************************************************************************/

BOOL WINAPI Template(LPSTR lpDIBBits, LONG lWidth, LONG lHeight,
                     int iTempH, int iTempW,
                     int iTempMX, int iTempMY,
                     FLOAT * fpArray, FLOAT fCoef)
```

```
{

    // 指向复制图像的指针
    LPSTR   lpNewDIBBits;

    HLOCAL  hNewDIBBits;

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 指向要复制区域的指针
    unsigned char*  lpDst;

    // 循环变量
    LONG    i;
    LONG    j;
    LONG    k;
    LONG    l;

    // 计算结果
    FLOAT   fResult;

    // 图像每行的字节数
    LONG lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lLineBytes * lHeight);

    // 判断是否内存分配失败
    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化图像为原始图像
    memcpy(lpNewDIBBits, lpDIBBits, lLineBytes * lHeight);

    // 行(除去边缘几行)
    for(i = iTempMY; i < lHeight - iTempH + iTempMY + 1; i++)
    {

        // 列(除去边缘几列)
        for(j = iTempMX; j < lWidth - iTempW + iTempMX + 1; j++)
        {
```

```
            // 指向新DIB第i行，第j个像素的指针
            lpDst = (unsigned char*)lpNewDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            fResult = 0;

            // 计算
            for (k = 0; k < iTempH; k++)
            {
                for (l = 0; l < iTempW; l++)
                {
                    // 指向DIB第i - iTempMY + k行，第j - iTempMX + l个像素的指针
                    lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i + iTempMY
                          - k) + j - iTempMX + l;

                    // 保存像素值
                    fResult += (* lpSrc) * fpArray[k * iTempW + l];
                }
            }

            // 乘上系数
            fResult *= fCoef;

            // 取绝对值
            fResult = (FLOAT ) fabs(fResult);

            // 判断是否超过255
            if(fResult > 255)
            {
                // 直接赋值为255
                * lpDst = 255;
            }
            else
            {
                // 赋值
                * lpDst = (unsigned char) (fResult + 0.5);
            }

        }
    }

    // 复制变换后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lLineBytes * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;

}
```

6.2.2 图像平滑理论基础

平滑模板的思想是通过一点和周围几个点的运算（通常为平均运算）来去除突然变化的点，从而滤掉一定的噪声，但图像有一定程度的模糊。而减少图像模糊代价是图像平滑研究的主要问题之一。这主要就取决于噪声本身的特性。一般情况下通过选择不同的模板来消除不同的噪声。

常用的模板有：

$$\frac{1}{4}\begin{bmatrix}0 & 1 & 0\\1 & 1\bullet & 1\\0 & 1 & 0\end{bmatrix}\quad \frac{1}{8}\begin{bmatrix}1 & 1 & 1\\1 & 0\bullet & 1\\1 & 1 & 1\end{bmatrix}\quad \frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1\bullet & 1\\1 & 1 & 1\end{bmatrix}$$

$$\frac{1}{10}\begin{bmatrix}1 & 1 & 1\\1 & 2\bullet & 1\\1 & 1 & 1\end{bmatrix}\quad \frac{1}{16}\begin{bmatrix}1 & 2 & 1\\2 & 4\bullet & 2\\1 & 2 & 1\end{bmatrix}$$

其中最后一个模板又常称为高斯模板，它是通过采样 2 维高斯函数得到的。

6.2.3 Visual C++编程实现

下面继续完善我们的图像处理应用程序。首先在菜单中添加一个“图像增强”菜单。如图 6－2 所示。

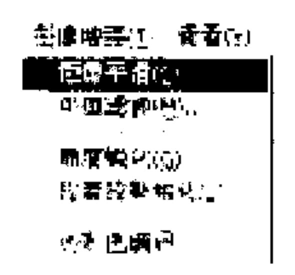


图 6－2 图像增强菜单

在图像平滑菜单项的单击事件中添加如下代码：

```
void CCh1_1View::OnEnhaSmooth()
{
    // 图像平滑

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;
```

```
// 模板高度
int     iTempH;

// 模板宽度
int     iTempW;

// 模板系数
FLOAT   fTempC;

// 模板中心元素X坐标
int     iTempMX;

// 模板中心元素Y坐标
int     iTempMY;

// 模板元素数组
FLOAT   aValue[25];

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的平滑，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的平滑！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 参数对话框
CDlgSmooth dlgPara;

// 给模板数组赋初值（为平均模板）
aValue[0] = 1.0;
aValue[1] = 1.0;
aValue[2] = 1.0;
aValue[3] = 0.0;
aValue[4] = 0.0;
aValue[5] = 1.0;
aValue[6] = 1.0;
aValue[7] = 1.0;
aValue[8] = 0.0;
```

```
aValue[9] = 0.0;
aValue[10] = 1.0;
aValue[11] = 1.0;
aValue[12] = 1.0;
aValue[13] = 0.0;
aValue[14] = 0.0;
aValue[15] = 0.0;
aValue[16] = 0.0;
aValue[17] = 0.0;
aValue[18] = 0.0;
aValue[19] = 0.0;
aValue[20] = 0.0;
aValue[21] = 0.0;
aValue[22] = 0.0;
aValue[23] = 0.0;
aValue[24] = 0.0;

// 初始化对话框变量值
dlgPara.m_intType = 0;
dlgPara.m_iTempH  = 3;
dlgPara.m_iTempW  = 3;
dlgPara.m_iTempMX = 1;
dlgPara.m_iTempMY = 1;
dlgPara.m_fTempC  = (FLOAT) (1.0 / 9.0);
dlgPara.m_fpArray = aValue;

// 显示对话框，提示用户设定平移量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户设定的平移量
iTempH  = dlgPara.m_iTempH;
iTempW  = dlgPara.m_iTempW;
iTempMX = dlgPara.m_iTempMX;
iTempMY = dlgPara.m_iTempMY;
fTempC  = dlgPara.m_fTempC;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 调用Template()函数平滑DIB
if (::Template(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB),
     iTempH, iTempW, iTempMX, iTempMY, aValue, fTempC))
{
```

```
        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败!", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
}
```

代码中 CdlgSmooth 是一个新创建的对话框类，该对话框主要是让用户设置平滑模板。它的完整代码如下：

1. 对话框头文件

```
#if !defined(AFX_DLGSMOOTH_H__DA1CA811_9B09_49C3_9598_E62B2757D073__INCLUDED_)
#define AFX_DLGSMOOTH_H__DA1CA811_9B09_49C3_9598_E62B2757D073__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgSmooth.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgSmooth dialog

class CDlgSmooth : public CDialog
{
// Construction
public:
    void UpdateEdit();

    // 模板元素数组指针
    FLOAT * m_fpArray;

    CDlgSmooth(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgSmooth)
    enum { IDD = IDD_DLG_Smooth };

    // 模板类型
```

```
	int	m_intType;

	// 模板高度
	int	m_iTempH;

	// 模板宽度
	int	m_iTempW;

	// 模板中心元素X坐标
	int	m_iTempMX;

	// 模板中心元素Y坐标
	int	m_iTempMY;

	// 模板系数
	float	m_fTempC;

	//}}AFX_DATA


// Overrides
	// ClassWizard generated virtual function overrides
	//{{AFX_VIRTUAL(CDlgSmooth)
	protected:
	virtual void DoDataExchange(CDataExchange* pDX);	// DDX/DDV support
	//}}AFX_VIRTUAL

// Implementation
protected:

	// Generated message map functions
	//{{AFX_MSG(CDlgSmooth)
	afx_msg void OnRad1();
	afx_msg void OnRad2();
	afx_msg void OnRad3();
	afx_msg void OnChangeEditTempw();
	virtual void OnOK();
	afx_msg void OnKillfocusEditTemph();
	afx_msg void OnKillfocusEditTempw();
	//}}AFX_MSG
	DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_DLGSMOOTH_H__DA1CA811_9B09_49C3_9598_E62B2757D073__INCLUDED_)
```

2. 对话框代码

```
// DlgSmooth.cpp : implementation file
```

```
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgSmooth.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgSmooth dialog


CDlgSmooth::CDlgSmooth(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgSmooth::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgSmooth)
    m_intType = -1;
    m_iTempH = 0;
    m_iTempW = 0;
    m_iTempMX = 0;
    m_iTempMY = 0;
    m_fTempC = 0.0f;
    //}}AFX_DATA_INIT
}


void CDlgSmooth::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgSmooth)
    DDX_Radio(pDX, IDC_RAD1, m_intType);
    DDX_Text(pDX, IDC_EDIT_TEMPH, m_iTempH);
    DDV_MinMaxInt(pDX, m_iTempH, 2, 5);
    DDX_Text(pDX, IDC_EDIT_TEMPW, m_iTempW);
    DDV_MinMaxInt(pDX, m_iTempW, 2, 5);
    DDX_Text(pDX, IDC_EDIT_MX, m_iTempMX);
    DDX_Text(pDX, IDC_EDIT_MY, m_iTempMY);
    DDX_Text(pDX, IDC_EDIT_TEMPC, m_fTempC);
    DDX_Text(pDX, IDC_EDIT_V0, m_fpArray[0]);
    DDX_Text(pDX, IDC_EDIT_V1, m_fpArray[1]);
    DDX_Text(pDX, IDC_EDIT_V2, m_fpArray[2]);
    DDX_Text(pDX, IDC_EDIT_V3, m_fpArray[3]);
    DDX_Text(pDX, IDC_EDIT_V4, m_fpArray[4]);
    DDX_Text(pDX, IDC_EDIT_V5, m_fpArray[5]);
    DDX_Text(pDX, IDC_EDIT_V6, m_fpArray[6]);
    DDX_Text(pDX, IDC_EDIT_V7, m_fpArray[7]);
    DDX_Text(pDX, IDC_EDIT_V8, m_fpArray[8]);
```

```
    DDX_Text(pDX, IDC_EDIT_V9, m_fpArray[9]);
    DDX_Text(pDX, IDC_EDIT_V10, m_fpArray[10]);
    DDX_Text(pDX, IDC_EDIT_V11, m_fpArray[11]);
    DDX_Text(pDX, IDC_EDIT_V12, m_fpArray[12]);
    DDX_Text(pDX, IDC_EDIT_V13, m_fpArray[13]);
    DDX_Text(pDX, IDC_EDIT_V14, m_fpArray[14]);
    DDX_Text(pDX, IDC_EDIT_V15, m_fpArray[15]);
    DDX_Text(pDX, IDC_EDIT_V16, m_fpArray[16]);
    DDX_Text(pDX, IDC_EDIT_V17, m_fpArray[17]);
    DDX_Text(pDX, IDC_EDIT_V18, m_fpArray[18]);
    DDX_Text(pDX, IDC_EDIT_V19, m_fpArray[19]);
    DDX_Text(pDX, IDC_EDIT_V20, m_fpArray[20]);
    DDX_Text(pDX, IDC_EDIT_V21, m_fpArray[21]);
    DDX_Text(pDX, IDC_EDIT_V22, m_fpArray[22]);
    DDX_Text(pDX, IDC_EDIT_V23, m_fpArray[23]);
    DDX_Text(pDX, IDC_EDIT_V24, m_fpArray[24]);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgSmooth, CDialog)
    //{{AFX_MSG_MAP(CDlgSmooth)
    ON_BN_CLICKED(IDC_RAD1, OnRad1)
    ON_BN_CLICKED(IDC_RAD2, OnRad2)
    ON_BN_CLICKED(IDC_RAD3, OnRad3)
    ON_EN_KILLFOCUS(IDC_EDIT_TEMPH, OnKillfocusEditTemph)
    ON_EN_KILLFOCUS(IDC_EDIT_TEMPW, OnKillfocusEditTempw)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgSmooth message handlers

void CDlgSmooth::OnRad1()
{
    // 3×3平均模板
    m_intType = 0;
    m_iTempH  = 3;
    m_iTempW  = 3;
    m_iTempMX = 1;
    m_iTempMY = 1;
    m_fTempC  = (FLOAT) (1.0 / 9.0);

    // 设置模板元素
    m_fpArray[0] = 1.0;
    m_fpArray[1] = 1.0;
    m_fpArray[2] = 1.0;
    m_fpArray[3] = 0.0;
    m_fpArray[4] = 0.0;
    m_fpArray[5] = 1.0;
    m_fpArray[6] = 1.0;
```

```
    m_fpArray[7] = 1.0;
    m_fpArray[8] = 0.0;
    m_fpArray[9] = 0.0;
    m_fpArray[10] = 1.0;
    m_fpArray[11] = 1.0;
    m_fpArray[12] = 1.0;
    m_fpArray[13] = 0.0;
    m_fpArray[14] = 0.0;
    m_fpArray[15] = 0.0;
    m_fpArray[16] = 0.0;
    m_fpArray[17] = 0.0;
    m_fpArray[18] = 0.0;
    m_fpArray[19] = 0.0;
    m_fpArray[20] = 0.0;
    m_fpArray[21] = 0.0;
    m_fpArray[22] = 0.0;
    m_fpArray[23] = 0.0;
    m_fpArray[24] = 0.0;

    // 更新文本框状态
    UpdateEdit();

    // 更新
    UpdateData(FALSE);

}

void CDlgSmooth::OnRad2()
{
    // 3×3高斯模板
    m_intType = 1;
    m_iTempH  = 3;
    m_iTempW  = 3;
    m_iTempMX = 1;
    m_iTempMY = 1;
    m_fTempC  = (FLOAT) (1.0 / 16.0);

    // 设置模板元素
    m_fpArray[0] = 1.0;
    m_fpArray[1] = 2.0;
    m_fpArray[2] = 1.0;
    m_fpArray[3] = 0.0;
    m_fpArray[4] = 0.0;
    m_fpArray[5] = 2.0;
    m_fpArray[6] = 4.0;
    m_fpArray[7] = 2.0;
    m_fpArray[8] = 0.0;
    m_fpArray[9] = 0.0;
    m_fpArray[10] = 1.0;
    m_fpArray[11] = 2.0;
    m_fpArray[12] = 1.0;
```

```
    m_fpArray[13] = 0.0;
    m_fpArray[14] = 0.0;
    m_fpArray[15] = 0.0;
    m_fpArray[16] = 0.0;
    m_fpArray[17] = 0.0;
    m_fpArray[18] = 0.0;
    m_fpArray[19] = 0.0;
    m_fpArray[20] = 0.0;
    m_fpArray[21] = 0.0;
    m_fpArray[22] = 0.0;
    m_fpArray[23] = 0.0;
    m_fpArray[24] = 0.0;

    // 更新文本框状态
    UpdateEdit();

    // 更新
    UpdateData(FALSE);

}

void CDlgSmooth::OnRad3()
{
    // 自定义模板
    m_intType = 2;

    // 更新文本框状态
    UpdateEdit();

}

void CDlgSmooth::OnOK()
{
    // 获取用户设置（更新）
    UpdateData(TRUE);

    // 判断设置是否有效
    if ((m_iTempMX < 0) || (m_iTempMX > m_iTempW - 1) ||
        (m_iTempMY < 0) || (m_iTempMY > m_iTempH - 1))
    {
        // 提示用户参数设置错误
        MessageBox("中心元素参数设置错误！", "系统提示",
            MB_ICONINFORMATION | MB_OK);

        // 返回
        return;
    }

    // 更新模板元素数组（将有效元素放置在数组的前面）
    for (int i = 0; i < m_iTempH; i++)
    {
```

```
        for (int j = 0; j < m_iTempW; j++)
        {
            m_fpArray[i * m_iTempW + j] = m_fpArray[i * 5 + j];
        }
    }

    // 更新
    UpdateData(FALSE);

    // 退出
    CDialog::OnOK();
}

void CDlgSmooth::UpdateEdit()
{
    BOOL    bEnable;

    // 循环变量
    int     i;
    int     j;

    // 判断是不是自定义模板
    if (m_intType == 2)
    {
        bEnable = TRUE;
    }
    else
    {
        bEnable = FALSE;
    }

    // 设置文本框可用状态
    (CEdit *) GetDlgItem(IDC_EDIT_TEMPH)->EnableWindow(bEnable);
    (CEdit *) GetDlgItem(IDC_EDIT_TEMPW)->EnableWindow(bEnable);
    (CEdit *) GetDlgItem(IDC_EDIT_TEMPC)->EnableWindow(bEnable);
    (CEdit *) GetDlgItem(IDC_EDIT_MX)->EnableWindow(bEnable);
    (CEdit *) GetDlgItem(IDC_EDIT_MY)->EnableWindow(bEnable);

    // IDC_EDIT_V0等ID其实是一个整数，它的数值定义在Resource.h中定义。

    // 设置模板元素文本框Enable状态
    for (i = IDC_EDIT_V0; i <= IDC_EDIT_V24; i++)
    {
        // 设置文本框不可编辑
        (CEdit *) GetDlgItem(i)->EnableWindow(bEnable);
    }

    // 显示应该可见的模板元素文本框
    for (i = 0; i < m_iTempH; i++)
    {
        for (j = 0; j < m_iTempW; j++)
```

```
        {
            // 设置文本框可见
            (CEdit *) GetDlgItem(IDC_EDIT_V0 + i*5 + j)->ShowWindow(SW_SHOW);
        }
    }

    // 隐藏应该不可见的模板元素文本框（前m_iTempH行的后几列）
    for (i = 0; i < m_iTempH; i++)
    {
        for (j = m_iTempW; j < 5; j++)
        {
            // 设置不可见
            (CEdit *) GetDlgItem(IDC_EDIT_V0 + i*5 + j)->ShowWindow(SW_HIDE);
        }
    }

    // 隐藏应该不可见的模板元素文本框（后几行）
    for (i = m_iTempH; i < 5; i++)
    {
        for (j = 0; j < 5; j++)
        {
            // 设置不可见
            (CEdit *) GetDlgItem(IDC_EDIT_V0 + i*5 + j)->ShowWindow(SW_HIDE);
        }
    }

}

void CDlgSmooth::OnKillfocusEditTemph()
{
    // 更新
    UpdateData(TRUE);

    // 更新文本框状态
    UpdateEdit();
}

void CDlgSmooth::OnKillfocusEditTempw()
{
    // 更新
    UpdateData(TRUE);

    // 更新文本框状态
    UpdateEdit();
}
```

原图如图 6－3 所示，利用上面代码用平均模板$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\1 & 1\bullet & 1\\1 & 1 & 1\end{bmatrix}$处理后的结果如图 6－4 所

示，用高斯模板处理后的结果如图 6－5 所示。

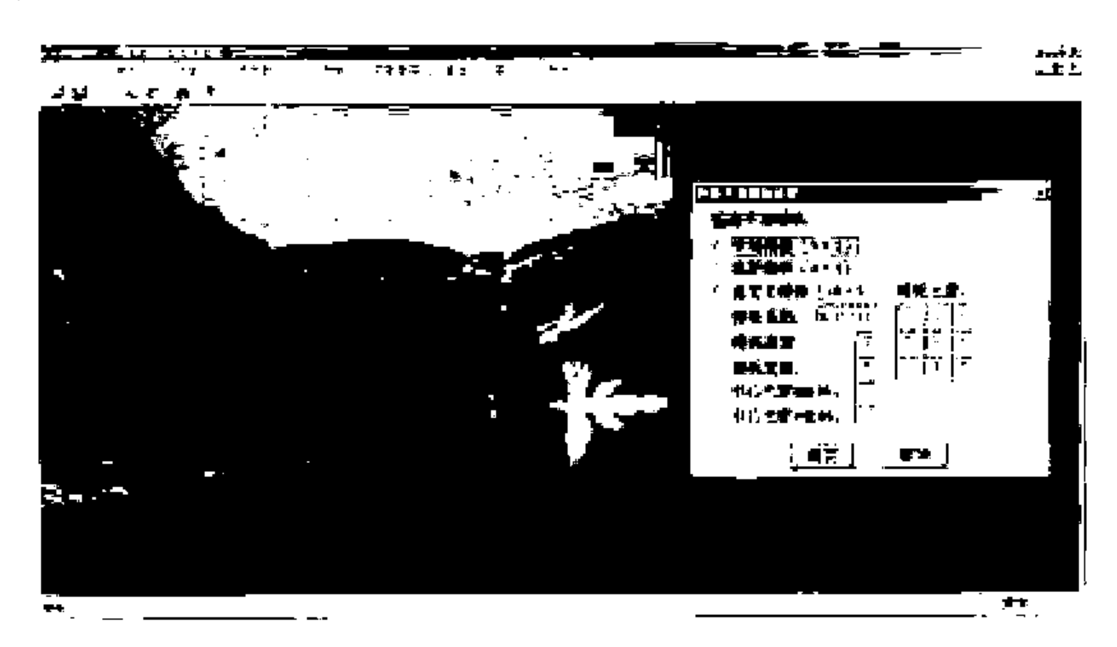




图 6－4 利用平均模板处理后的图像

上面的代码其实可以实现任何<=5×5 的模板操作。如图 6－3 中的对话框所示，选择“自定义模板”，这样就可以自己定义模板的系数、模板的高度和宽度、模板中心元素的位置、以及模板元素的取值。其实通过模板，我们还可以实现图像的平移、边缘识别。如果设模板



图 6－5 利用高斯模板处理后的图像

为：$\begin{bmatrix} 0\bullet & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$，则运行结果应该是图像向左上移动 5 个像素。运行结果如图 6－6 所示（注意与原图 6－3 对比，5 个像素比较小，效果可能不是很明显）。

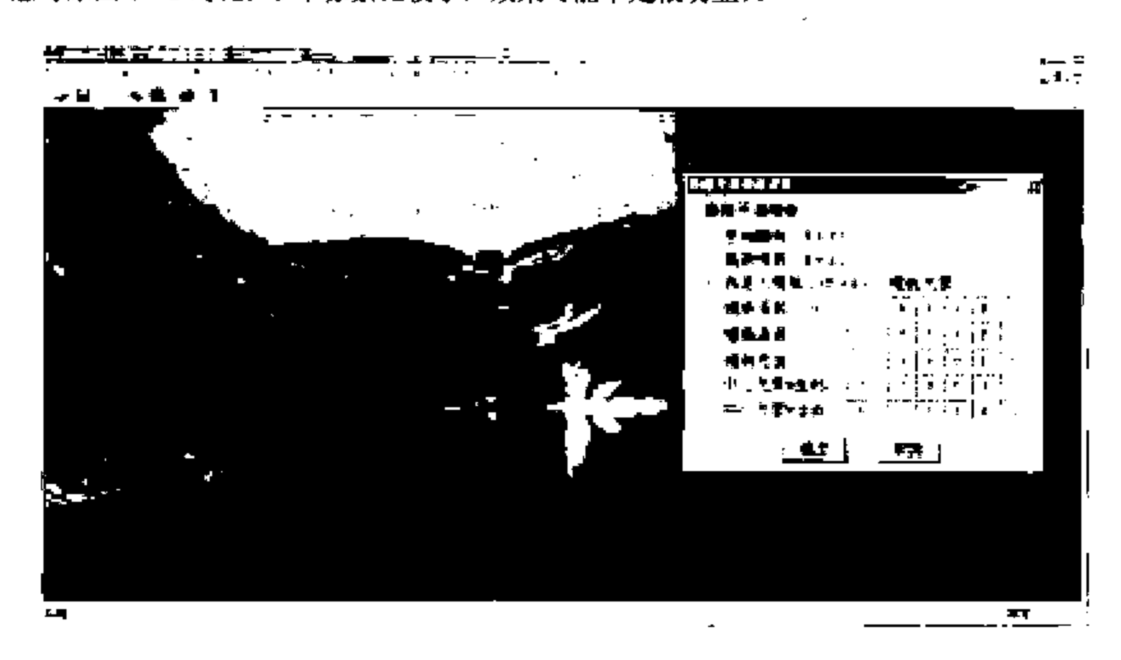


图 6－6 利用模板操作平移图像

如果取模板为$\begin{bmatrix}0 & 1 & 0\\ 1 & -4\bullet & 1\\ 0 & 1 & 0\end{bmatrix}$，则可以实现图像边缘的识别。如图6－7和图6－8所示。

This is a sample.

这是一个示例。

图6－7　原始图像图

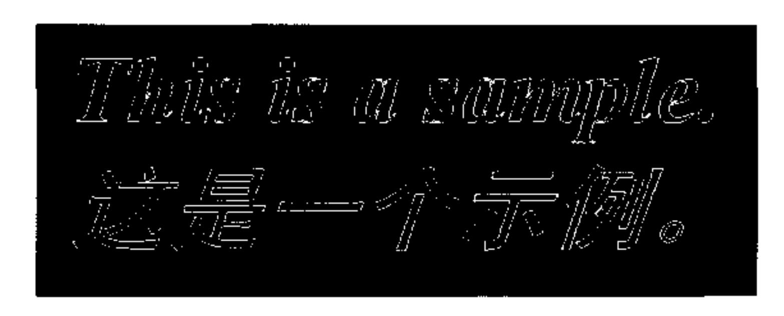


图6－8　模板处理后的图像

处理后图像中灰度值非0处即为图像的边界。它的原理将在后面的章节中详细描述。

6.3　中值滤波

中值滤波是一种非线性的信号处理方法，与其对应的中值滤波器当然也就是一种非线性的滤波器。中值滤波器在1971年由J. w. Jukey首先提出并应用在一维信号处理技术（时间序列分析）中，后来被二维图像信号处理技术所引用。中值滤波在一定的条件下可以克服线性滤波器如最小均方滤波，均值滤波等带来的图像细节模糊，而且对滤除脉冲干扰及图像扫描噪声最为有效。由于在实际运算过程中不需要图像的统计特征，因此这也带来不少方便。但是对于一些细节多，特别是点、线、尖顶细节多的图像不宜采用中值滤波。

6.3.1　理论基础

中值滤波一般采用一个含有奇数个点的滑动窗口，将窗口中各点灰度值的中值来替代指定点（一般是窗口的中心点）的灰度值。对于奇数个元素，中值是指按大小排序后，中间的数值；对于偶数个元素，中值是指排序后中间两个元素灰度值的平均值。

对于一维情况，如图6－9所示。它是用内含5个元素（1×5）的窗口对离散阶跃函数、斜坡函数、脉冲函数以及三角形函数进行中值滤波和均值滤波的示例。

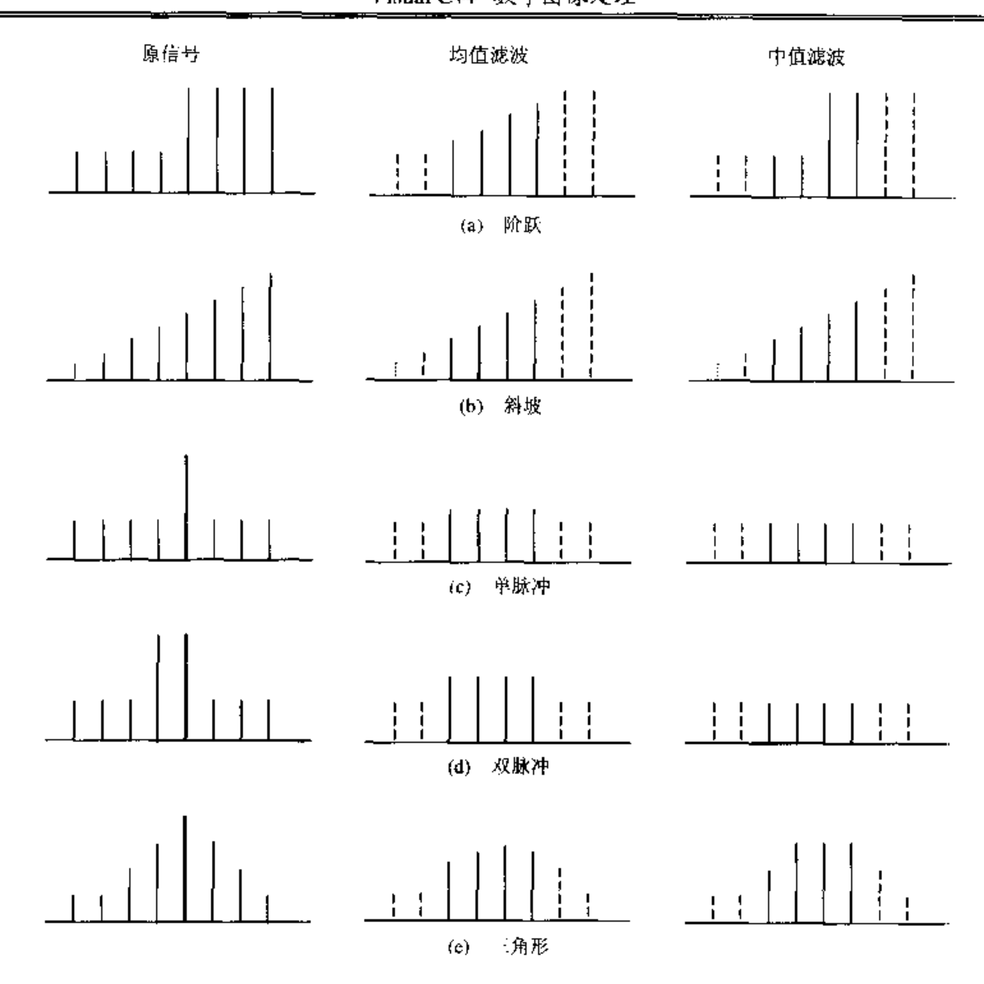


图 6—9 中值滤波和均值滤波比较示例

从该图中可以看出，在一维情况下，中值滤波器不影响阶跃函数和斜坡函数，并可以有效地消除单、双脉冲，使三角函数的顶端变平。

对于二维情况，中值滤波的窗口形状和尺寸对滤波器效果影响很大。不同图像内容和不同应用要求往往选用不同的窗口形状和尺寸。常用的二维中值滤波窗口形状有线状、方形、圆形、十字形等。在本节的编程实践中，我们只考虑方形一种情况的中值滤波，其他形状的中值滤波也可以由此情况类似完成。

6.3.2 Visual C++编程实现

下面我们来完成图像中值滤波函数的编写。完成中值滤波函数的编写，需要传递给子函数图像像素的指针和图像的高宽信息，而且还要传递滤波器的信息：滤波器的高宽和中心元素的位置。下面的代码实现了图像中值滤波功能。

```
/*************************************************************************
 *
 * 函数名称:
 *   MedianFilter()
 *
```

```
 * 参数:
 *   LPSTR lpDIBBits          - 指向原DIB图像指针
 *   LONG  lWidth             - 原图像宽度（像素数）
 *   LONG  lHeight            - 原图像高度（像素数）
 *   int   iFilterH           - 滤波器的高度
 *   int   iFilterW           - 滤波器的宽度
 *   int   iFilterMX          - 滤波器的中心元素X坐标
 *   int   iFilterMY          - 滤波器的中心元素Y坐标
 *
 * 返回值:
 *   BOOL                     - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数对DIB图像进行中值滤波。
 *
 ************************************************************************/

BOOL WINAPI MedianFilter(LPSTR lpDIBBits, LONG lWidth, LONG lHeight,
                         int iFilterH, int iFilterW,
                         int iFilterMX, int iFilterMY)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;

    // 指向要复制区域的指针
    unsigned char*  lpDst;

    // 指向复制图像的指针
    LPSTR           lpNewDIBBits;
    HLOCAL          hNewDIBBits;

    // 指向滤波器数组的指针
    unsigned char   * aValue;
    HLOCAL          hArray;

    // 循环变量
    LONG            i;
    LONG            j;
    LONG            k;
    LONG            l;

    // 图像每行的字节数
    LONG            lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lLineBytes * lHeight);
```

```
// 判断是否内存分配失败
if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化图像为原始图像
memcpy(lpNewDIBBits, lpDIBBits, lLineBytes * lHeight);

// 暂时分配内存，以保存滤波器数组
hArray = LocalAlloc(LHND, iFilterH * iFilterW);

// 判断是否内存分配失败
if (hArray == NULL)
{
    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 分配内存失败
    return FALSE;
}

// 锁定内存
aValue = (unsigned char * )LocalLock(hArray);

// 开始中值滤波
// 行(除去边缘几行)
for(i = iFilterMY; i < lHeight - iFilterH + iFilterMY + 1; i++)
{
    // 列(除去边缘几列)
    for(j = iFilterMX; j < lWidth - iFilterW + iFilterMX + 1; j++)
    {
        // 指向新DIB第i行，第j个像素的指针
        lpDst = (unsigned char*)lpNewDIBBits + lLineBytes * (lHeight - 1 - i) + j;

        // 读取滤波器数组
        for (k = 0; k < iFilterH; k++)
        {
            for (l = 0; l < iFilterW; l++)
            {
                // 指向DIB第i - iFilterMY + k行，第j - iFilterMX + l个像素的指针
                lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i +
                        iFilterMY - k) + j - iFilterMX + l;

                // 保存像素值
                aValue[k * iFilterW + l] = *lpSrc;
```

```
                }
            }

            // 获取中值
            * lpDst = GetMedianNum(aValue, iFilterH * iFilterW);
        }
    }

    // 复制变换后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lLineBytes * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);
    LocalUnlock(hArray);
    LocalFree(hArray);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   GetMedianNum()
 *
 * 参数:
 *   unsigned char * bpArray    - 指向要获取中值的数组指针
 *   int   iFilterLen           - 数组长度
 *
 * 返回值:
 *   unsigned char      - 返回指定数组的中值。
 *
 * 说明:
 *   该函数用冒泡法对一维数组进行排序，并返回数组元素的中值。
 *
 ************************************************************************/

unsigned char WINAPI GetMedianNum(unsigned char * bArray, int iFilterLen)
{
    // 循环变量
    int     i;
    int     j;

    // 中间变量
    unsigned char bTemp;

    // 用冒泡法对数组进行排序
    for (j = 0; j < iFilterLen - 1; j ++)
    {
        for (i = 0; i < iFilterLen - j - 1; i ++)
```

```
        {
            if (bArray[i] > bArray[i - 1])
            {
                // 互换
                bTemp = bArray[i];
                bArray[i] = bArray[i + 1];
                bArray[i + 1] = bTemp;
            }
        }
    }

    // 计算中值
    if ((iFilterLen & 1) > 0)
    {
        // 数组有奇数个元素，返回中间一个元素
        bTemp = bArray[(iFilterLen + 1) / 2];
    }
    else
    {
        // 数组有偶数个元素，返回中间两个元素平均值
        bTemp = (bArray[iFilterLen / 2] + bArray[iFilterLen / 2 + 1]) / 2;
    }

    // 返回中值
    return bTemp;
}
```

接下来编写中值滤波菜单事件的处理代码：

```
void CCh1_1View::OnENHAMidianF()
{
    // 中值滤波

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 滤波器的高度
    int iFilterH;

    // 滤波器的宽度
    int iFilterW;

    // 中心元素的X坐标
    int iFilterMX;

    // 中心元素的Y坐标
```

```
int iFilterMY;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的中值滤波，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的中值滤波！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 创建对话框
CDlgMidFilter dlgPara;

// 初始化变量值
dlgPara.m_iFilterType = 0;
dlgPara.m_iFilterH = 3;
dlgPara.m_iFilterW = 1;
dlgPara.m_iFilterMX = 0;
dlgPara.m_iFilterMY = 1;

// 显示对话框，提示用户设定平移量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户的设定
iFilterH = dlgPara.m_iFilterH;
iFilterW = dlgPara.m_iFilterW;
iFilterMX = dlgPara.m_iFilterMX;
iFilterMY = dlgPara.m_iFilterMY;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();
```

```
    // 调用MedianFilter()函数中值滤波
    if (::MedianFilter(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB),
          iFilterH, iFilterW, iFilterMX, iFilterMY))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

其中 CdlgMidFilter 类是一个新建的对话框类，它是用来获取用户设定的中值滤波器大小和中心元素位置的。完整代码如下所示：

1. 对话框头文件

```
#if !defined(AFX_DLGMIDFILTER_H__3844F7C0_0F6D_488D_97CE_1DF381742683__INCLUDED_)
#define AFX_DLGMIDFILTER_H__3844F7C0_0F6D_488D_97CE_1DF381742683__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgMidFilter.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgMidFilter dialog

class CDlgMidFilter : public CDialog
{
// Construction
public:
    CDlgMidFilter(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgMidFilter)
    enum { IDD = IDD_DLG_MidianFilter };
```

```
    // 滤波器类型
    int     m_iFilterType;

    // 滤波器高度
    int     m_iFilterH;

    // 滤波器宽度
    int     m_iFilterW;

    // 滤波器中心元素X坐标
    int     m_iFilterMX;

    // 滤波器中心元素Y坐标
    int     m_iFilterMY;

    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgMidFilter)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgMidFilter)
    afx_msg void OnRad1();
    afx_msg void OnRad2();
    afx_msg void OnRad3();
    afx_msg void OnRad4();
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C-- will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_DLGMIDFILTER_H__3844F7C0_0F6D_488D_97CE_1DF381742683__INCLUDED_)
```

2. 对话框代码

```
// DlgMidFilter.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
```

```
#include "DlgMidFilter.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgMidFilter dialog


CDlgMidFilter::CDlgMidFilter(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgMidFilter::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgMidFilter)
    m_iFilterType = -1;
    m_iFilterH = 0;
    m_iFilterMX = 0;
    m_iFilterMY = 0;
    m_iFilterW = 0;
    //}}AFX_DATA_INIT
}


void CDlgMidFilter::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgMidFilter)
    DDX_Radio(pDX, IDC_RAD1, m_iFilterType);
    DDX_Text(pDX, IDC_EDIT_FH, m_iFilterH);
    DDV_MinMaxInt(pDX, m_iFilterH, 1, 8);
    DDX_Text(pDX, IDC_EDIT_FMX, m_iFilterMX);
    DDX_Text(pDX, IDC_EDIT_FMY, m_iFilterMY);
    DDX_Text(pDX, IDC_EDIT_FW, m_iFilterW);
    DDV_MinMaxInt(pDX, m_iFilterW, 1, 8);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgMidFilter, CDialog)
    //{{AFX_MSG_MAP(CDlgMidFilter)
    ON_BN_CLICKED(IDC_RAD1, OnRad1)
    ON_BN_CLICKED(IDC_RAD2, OnRad2)
    ON_BN_CLICKED(IDC_RAD3, OnRad3)
    ON_BN_CLICKED(IDC_RAD4, OnRad4)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgMidFilter message handlers
```

```
void CDlgMidFilter::OnRad1()
{
    // 3×1模板
    m_iFilterType = 0;
    m_iFilterH = 3;
    m_iFilterW = 1;
    m_iFilterMX = 0;
    m_iFilterMY = 1;
    // 设置文本框不可用
    (CEdit *) GetDlgItem(IDC_EDIT_FH)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FW)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMX)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMY)->EnableWindow(FALSE);
    // 更新
    UpdateData(FALSE);
}

void CDlgMidFilter::OnRad2()
{
    // 1×3模板
    m_iFilterType = 1;
    m_iFilterH = 1;
    m_iFilterW = 3;
    m_iFilterMX = 1;
    m_iFilterMY = 0;
    // 设置文本框不可用
    (CEdit *) GetDlgItem(IDC_EDIT_FH)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FW)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMX)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMY)->EnableWindow(FALSE);
    // 更新
    UpdateData(FALSE);
}

void CDlgMidFilter::OnRad3()
{
    // 3×3模板
    m_iFilterType = 2;
    m_iFilterH = 3;
    m_iFilterW = 3;
    m_iFilterMX = 1;
    m_iFilterMY = 1;
    // 设置文本框不可用
    (CEdit *) GetDlgItem(IDC_EDIT_FH)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FW)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMX)->EnableWindow(FALSE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMY)->EnableWindow(FALSE);
    // 更新
    UpdateData(FALSE);
}
```

```
void CDlgMidFilter::OnRad4()
{
    // 自定义模板
    (CEdit *) GetDlgItem(IDC_EDIT_FH)->EnableWindow(TRUE);
    (CEdit *) GetDlgItem(IDC_EDIT_FW)->EnableWindow(TRUE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMX)->EnableWindow(TRUE);
    (CEdit *) GetDlgItem(IDC_EDIT_FMY)->EnableWindow(TRUE);
}

void CDlgMidFilter::OnOK()
{
    // 获取用户设置（更新）
    UpdateData(TRUE);

    // 判断设置是否有效
    if ((m_iFilterMX < 0) || (m_iFilterMX > m_iFilterW - 1) ||
        (m_iFilterMY < 0) || (m_iFilterMY > m_iFilterH - 1))
    {
        // 提示用户参数设置错误
        MessageBox("参数设置错误！", "系统提示", MB_ICONINFORMATION | MB_OK);

        // 返回
        return;
    }

    // 退出
    CDialog::OnOK();
}
```

利用上述代码，图 6－3 进行 3×3 中值滤波结果如图 6－10 所示。

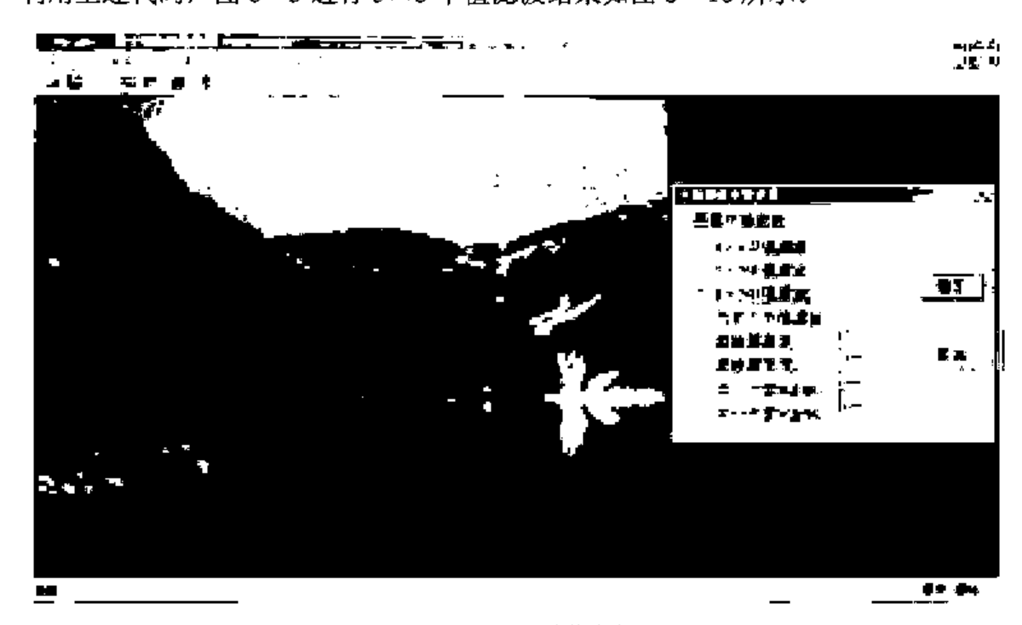


图 6－10　3×3 中值滤波

6.4 图像的锐化

图像锐化处理的目的是使模糊的图像变得更加清晰起来。通常针对引起图像模糊的原因而进行相应地锐化操作属于图像复原的内容，在这里只是介绍一般的去模糊算法。

图像的模糊实质就是图像受到平均或积分运算造成的，因此可以对图像进行逆运算如微分运算来使图像清晰化。从频谱角度来分析，图像模糊的实质是其高频分量被衰减，因而可以通过高通滤波操作来清晰图像。但要注意，能够进行锐化处理的图像必须有较高的信噪比，否则锐化后图像信噪比反而更低，从而使噪声的增加得比信号还要多，因此一般是先去除或减轻噪声后再进行锐化处理。

图像锐化一般有两种方法：一种是微分法，另外一种是高通滤波法。后者的工作原理和低通滤波相似，这里就不再详细介绍了。下面主要介绍一下两种常用的微分锐化方法：梯度锐化和拉普拉斯锐化。对于高通滤波法，只给出几种常用的高通滤波器。

6.4.1 梯度锐化

设图像为$f(x, y)$，定义$f(x, y)$在点(x, y)处的梯度矢量$\vec{G}[f(x, y)]$为：

$$\vec{G}[f(x,y)]=\begin{bmatrix}\dfrac{\partial f}{\partial x}\\ \dfrac{\partial f}{\partial y}\end{bmatrix}$$

梯度有两个重要的性质：

- 梯度的方向在函数$f(x, y)$最大变化率方向上。
- 梯度的幅度用$G[f(x,y)]$表示，其值为：

$$G[f(x,y)]=\sqrt{\left(\frac{\partial f}{\partial x}\right)^2+\left(\frac{\partial f}{\partial y}\right)^2}$$

由此式可得出这样的结论：梯度的数值就是$f(x, y)$在其最大变化率方向上的单位距离所增加的量。

对于离散的数字图像，上式可以改写成：

$$G[f(i,j)]=\sqrt{[f(i,j)-f(i+1,j)]^2+[f(i,j)-f(i,j+1)]^2}$$

为了计算方便，也可以采用下面的近似计算公式：

$$G[f(i,j)]\cong|f(i,j)-f(i+1,j)|+|f(i,j)-f(i,j+1)| \qquad (1)$$

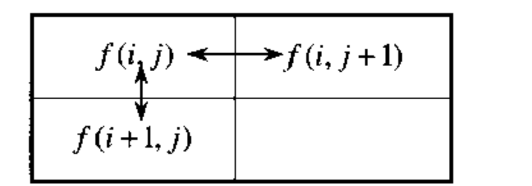
通常也可以近似为下面两种形式：

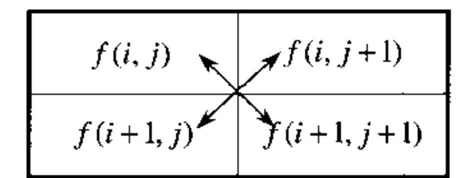
$$G[f(i,j)]=\sqrt{[f(i,j)-f(i+1,j+1)]^2+[f(i+1,j)-f(i,j+1)]^2}$$

$$G[f(i,j)] \cong |f(i,j)-f(i+1,j+1)|+|f(i+1,j)-f(i,j+1)| \qquad (2)$$

上面两个公式称为罗伯特（Roberts）梯度。该梯度定义在数学上也许没有道理，但是它运算简单、实用，而且效果也不错，通常在应用中也采用该梯度方式。

公式（1）和（2）示意图如图 6－11 所示。





6－11　计算公式示意图

如果直接采用梯度值$G[f(x,y)]$来表示图像，即令$g(x,y)=G[f(x,y)]$，则由上面的公式可见：在图像变化缓慢的地方其值很小（对应于图像较暗）；而在线条轮廓等变化较快的地方的值很大。这就是图像在经过梯度运算后使其清晰从而达到锐化的目的。

这和后面将要介绍的边缘检测算法有相通之处，其实罗伯特梯度算法本身就是一种常用的边缘检测算法。

由于在图像变化缓慢的地方梯度很小，所以图像会显得很暗，通常的做法是给一个阈值Δ，如果$G[f(x,y)]$小于该阈值Δ，则保持原灰度值不变；如果大于或等于阈值Δ，则赋值为$G[f(x,y)]$：

$$g(x,y)=\begin{cases}G[f(x,y)] & (G[f(x,y)]\ge\Delta)\\ f(x,y) & (G[f(x,y)]<\Delta)\end{cases} \qquad (3)$$

或者：

$$g(x,y)=\begin{cases}L_a & (G[f(x,y)]\ge\Delta)\\ f(x,y) & (G[f(x,y)]<\Delta)\end{cases}$$

其中 L_a 为一个固定的灰度值。同样也可以使图像有一个固定背景灰度值 L_b，以突出边缘灰度的变换。其变换公式如下：

$$g(x,y)=\begin{cases}G[f(x,y)] & (G[f(x,y)]\ge\Delta)\\ L_b & (G[f(x,y)]<\Delta)\end{cases}$$

甚至可以只保留两个灰度值，以供研究边缘位置：

$$g(x,y)=\begin{cases}L_a & (G[f(x,y)]\ge\Delta)\\ L_b & (G[f(x,y)]<\Delta)\end{cases}$$

下面我们来编写梯度锐化的 API 函数 GradSharp()。在这里我们采用近似公式（1）和（3）

来进行梯度变换，函数中阈值Δ由参数bThre指定。

```
/*************************************************************************
 *
 * 函数名称:
 *   GradSharp()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth         原图像宽度（像素数）
 *   LONG  lHeight        原图像高度（像素数）
 *   BYTE  bThre        - 阈值
 *
 * 返回值:
 *   BOOL               - 成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对图像进行梯度锐化。
 *
 ************************************************************************/
BOOL WINAPI GradSharp(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BYTE bThre)
{

    // 指向原图像的指针
    unsigned char*  lpSrc;
    unsigned char*  lpSrc1;
    unsigned char*  lpSrc2;

    // 循环变量
    LONG    i;
    LONG    j;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 中间变量
    BYTE    bTemp;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc  = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 指向DIB第i+1行，第j个像素的指针
```

```
            lpSrc1 = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 2 - i) + j;

            // 指向DIB第i行，第j+1个像素的指针
            lpSrc2 = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j + 1;

            bTemp = abs((*lpSrc)-(*lpSrc1)) + abs((*lpSrc)-(*lpSrc2));

            // 判断是否小于阈值
            if (bTemp < 255)
            {
                // 判断是否大于阈值，对于小于情况，灰度值不变。
                if (bTemp >= bThre)
                {
                    // 直接赋值为bTemp
                    *lpSrc = bTemp;
                }
            }
            else
            {
                // 直接赋值为255
                *lpSrc = 255;
            }
        }
    }

    // 返回
    return TRUE;
}
```

对于其他几种梯度锐化方式，也可以通过类似方法来实现。下面我们来编写梯度锐化菜单处理事件的代码：

```
void CCh1_1View::OnEnhaGradsharp()
{
    // 梯度锐化

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的梯度锐化，其他的可以类推）
```

```
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的梯度锐化！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 阈值
BYTE    bThre;

// 创建对话框
CDlgSharpThre dlgPara;

// 初始化变量值
dlgPara.m_bThre = 10;

// 提示用户输入阈值
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户的设定
bThre = dlgPara.m_bThre;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 调用GradSharp()函数进行梯度板锐化
if (::GradSharp(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB), bThre))
{

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);
}
else
{
    // 提示用户
```

其中 CdlgSharpThre 为一个新创建的对话框类。该对话框比较简单，只有一个类成员变量 m_bThre 来保存用户设定的阈值，这里就不给出它的源代码了。

图 6－3 梯度锐化（阈值取 10）的结果如图 6－12 所示。

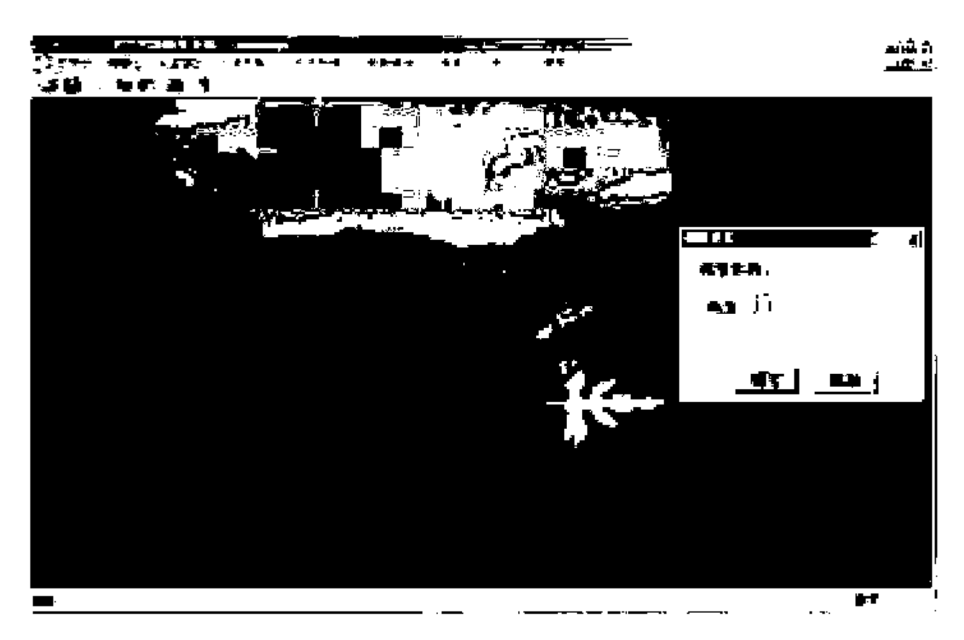


图 6－12　梯度锐化

如果取阈值为 0，来处理图 6－7，结果如图 6－13 所示。



图 6－13　梯度锐化进行边缘检测

可见也可以用梯度锐化来进行边缘检测。

6.4.2 拉普拉斯锐化

拉普拉斯运算也是偏导数运算的线性组合，而且是一种各向同性（旋转不变性）的线性运算。设$\nabla^2 f$为拉普拉斯算子，则：

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

对于离散数字图像 f (i, j)，其一阶偏导数为：

$$\begin{cases} \dfrac{\partial f(i,j)}{\partial x} = \Delta_x f(i,j) = f(i,j) - f(i-1,j) \\ \dfrac{\partial f(i,j)}{\partial y} = \Delta_y f(i,j) = f(i,j) - f(i,j-1) \end{cases}$$

则其二阶偏导数为：

$$\begin{cases} \dfrac{\partial^2 f(i,j)}{\partial x^2} = \Delta_x f(i+1,j) - \Delta_x f(i,j) = f(i+1,j) + f(i-1,j) - 2f(i,j) \\ \dfrac{\partial^2 f(i,j)}{\partial y^2} = \Delta_y f(i,j+1) - \Delta_y f(i,j) = f(i,j+1) + f(i,j-1) - 2f(i,j) \end{cases}$$

所以，拉普拉斯算子$\nabla^2 f$为：

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2} = f(i-1,j) + f(i+1,j) + f(i,j+1) + f(i,j-1) - 4f(i,j)$$

对于扩散现象引起的图像模糊，可以用下式来进行锐化：

$$g(i,j) = f(i,j) - k\tau\nabla^2 f(i,j)$$

这里$k\tau$是与扩散效应有关的系数。该系数取值要合理，如果$k\tau$过大，图像轮廓边缘会产生过冲；反之如果$k\tau$过小，锐化效果就不明显。

如果令$k\tau = 1$，则变换公式为：

$$g(i,j) = 5f(i,j) - f(i-1,j) - f(i+1,j) - f(i,j+1) - f(i,j-1)$$

用模板表示如下：

$$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5\bullet & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

这样拉普拉斯锐化运算完全可以转换成模板运算。

其实，我们通常用的拉普拉斯锐化模板还有另外一种形式：

$$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 9\bullet & -1 \\ -1 & -1 & -1 \end{bmatrix}$$

下面我们来用上述模板进行图像锐化，添加菜单拉普拉斯锐化处理代码：

```
void CCh1_1View::OnEnhaSharp()
{
    // 图像拉普拉斯锐化

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 模板高度
    int      iTempH;

    // 模板宽度
    int      iTempW;

    // 模板系数
    FLOAT    fTempC;

    // 模板中心元素X坐标
    int      iTempMX;

    // 模板中心元素Y坐标
    int      iTempMY;

    // 模板元素数组
    FLOAT    aValue[9];

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的锐化，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的锐化！", "系统提示",
            MB_ICONINFORMATION | MB_OK);
```

```
        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 设置拉普拉斯模板参数
    iTempW = 3;
    iTempH = 3;
    fTempC = 1.0;
    iTempMX = 1;
    iTempMY = 1;
    aValue[0] = -1.0;
    aValue[1] = -1.0;
    aValue[2] = -1.0;
    aValue[3] = -1.0;
    aValue[4] =  9.0;
    aValue[5] = -1.0;
    aValue[6] = -1.0;
    aValue[7] = -1.0;
    aValue[8] = -1.0;

    // 调用Template()函数用拉普拉斯模板锐化DIB
    if (::Template(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB),
          iTempH, iTempW, iTempMX, iTempMY, aValue, fTempC))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
```

运行上述代码锐化图6-3结果如图6-14所示。



图 6－14 拉普拉斯锐化

6.4.3 高通滤波器

由于图像中的边缘及急剧变化部分与图像高频分量有关，因此利用高通滤波器衰减图像信号的低频部分能相对增强图像高频部分，从而实现图像锐化的目的。常用的高通滤波器有理想高通滤波器、巴特沃夫高通滤波器、指数高通滤波器和梯形高通滤波器。下面将分别给出各种高通滤波器的转移函数。

1. 理想高通滤波器

理想二维高通滤波器的传递函数如下：

$$H(\mu,\nu)=\begin{cases}0 & D(\mu,\nu)\le D_0\\ 1 & D(\mu,\nu)>D_0\end{cases}$$

其中 D_0 是从频率平面原点算起的截止频率（或截止距离）。$D(\mu,\nu)$ 为：

$$D(\mu,\nu)=\sqrt{\mu^2+\nu^2}$$

理想高通滤波器传递函数的径向剖面图如图 6－15 所示。

理想高通滤波器和理想低通滤波器相反，它正好将以 D_0 为半径的圆内的频率成份（低频部分）衰减掉，而对圆外的频率成份（高频部分）则可以无损通过。

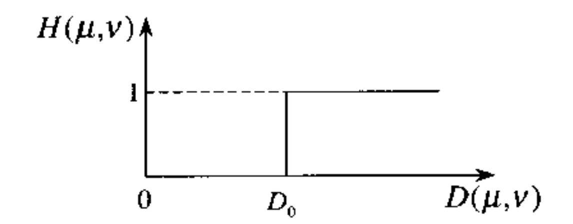


图 6－15 理想高通滤波器传递函数径向剖面图

2. 巴特沃夫高通滤波器

截止频率为 D_0 的 n 阶巴特沃夫高通滤波器的传递函数如下所示：

$$H(\mu,v)=\frac{1}{1+\left[\frac{D_0}{D(\mu,v)}\right]^{2n}}$$

其中 $D(\mu,v)$ 仍然为：

$$D(\mu,v)=\sqrt{\mu^2+v^2}$$

巴特沃夫高通滤波器传递函数径向剖面图如图 6－16 所示。

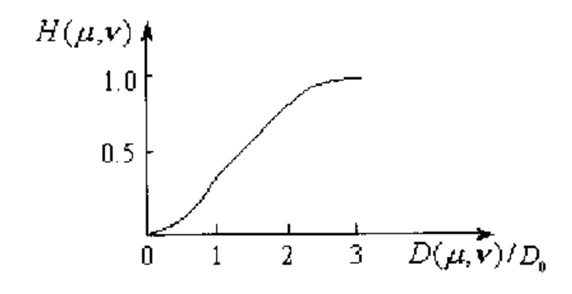


图 6－16 巴特沃夫高通滤波器传递函数径向剖面图

和低通滤波器类似，定义 $H(\mu,v)$ 下降到其最大值一半处的 $D(\mu,v)$ 为截频点 D_0。一般情况下，高通滤波器的截频选择使 $H(\mu,v)$ 下降到其最大值 $\frac{1}{\sqrt{2}}$ 处，满足该条件的传递函数可以修改为：

$$H(\mu,v)=\frac{1}{1+(\sqrt{2}-1)\left[\frac{D_0}{D(\mu,v)}\right]^{2n}}=\frac{1}{1+0.414\left[\frac{D_0}{D(\mu,v)}\right]^{2n}}$$

3. 指数高通滤波器

截频为 D_0 的指数高通滤波器的传递函数如下所示：

$$H(\mu,v)=\mathrm{e}^{-\left[\frac{D_0}{D(\mu,v)}\right]^n}$$

其中 D_0 为截频，$D(\mu,v)=\sqrt{\mu^2+v^2}$，参数 n 控制着 $H(\mu,v)$ 的增长率。指数高通滤波器的传递函数径向剖面图如图 6－17 所示。

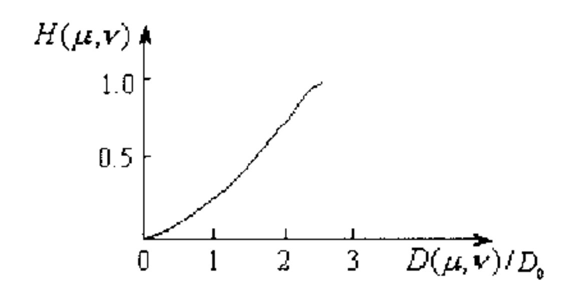


图 6－17　指数高通滤波器传递函数径向剖面图

当 $D(\mu,v)=D_0$ 时，$H(\mu,v)=\dfrac{1}{\mathrm{e}}$。如果仍然把截止频率定在 $H(\mu,v)$ 最大值 $\dfrac{1}{\sqrt{2}}$ 处，则传递函数可以修改为如下形式：

$$H(\mu,v)=\mathrm{e}^{\ln\frac{1}{\sqrt{2}}\left[\frac{D_0}{D(\mu,v)}\right]^n}=\mathrm{e}^{-0.347\left[\frac{D_0}{D(\mu,v)}\right]^n}$$

4. 梯形高通滤波器

梯形高通滤波器的传递函数可以用下式表示：

$$H(\mu,v)=\begin{cases}0 & D(\mu,v)<D_1\\ \dfrac{D(\mu,v)-D_1}{D_0-D_1} & D_1\le D(\mu,v)\le D_0\\ 1 & D(\mu,v)>D_0\end{cases}$$

同样式中 $D(\mu,v)=\sqrt{\mu^2+v^2}$。$D_0$ 和 D_1 为指定值，并且 $D_0>D_1$，定义截频为 D_0，D_1 是任意选的，只要满足 $D_0>D_1$ 即可。梯形高通滤波器的传递函数径向剖面图如图 6－18 所示。

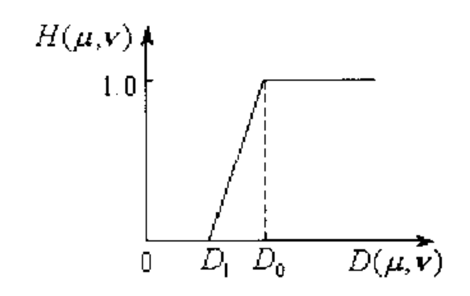


图 6-18 梯形高通滤波器传递函数径向剖面图

6.5 伪彩色编码

有时，对于一幅灰度图像，我们需要将它人为地转换成一幅彩色图像，这在医学图像处理中是很常见的。因为人眼对灰度微弱递变的敏感程度远远小于对色彩变化的敏感程度。因此，将一幅灰度图像按照特定的彩色编码表进行彩色变换，这样就可以看到图像更加精细的结构，以便于医师对疾病的诊断。

一般，要将灰度图进行伪彩色变换，可以采用一个 256 色的调色板（我们称之为伪彩色编码表），其中定义了每种灰度对应颜色的 RGB 值。

用 Visual C++来编程实现伪彩色变换非常方便，实际上就是用指定的伪彩色调色板来替换当前图像的调色板。完成该操作需要分二步：首先按照伪彩色编码表来更改当前 DIB 的调色板；其次接着调用调色板类 CPalette 的 SetPaletteEntries()函数来替换当前文档的调色板，并刷新当前视图实现变换。

下面我们首先编写一个通用的替换当前 DIB 调色板的函数 ReplaceColorPal()。该函数需要指向当前 DIB 的指针以及要替换的伪彩色编码表的指针。它的完整代码如下：

```
/*************************************************************************
 *
 * 函数名称：
 *   ReplaceColorPal()
 *
 * 参数：
 *   LPSTR lpDIB            - 指向原DIB图像指针
 *   BYTE * bpColorsTable   - 伪彩色编码表
 *
 * 返回值：
 *   BOOL                   - 成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用指定的伪彩色编码表来替换图像的调试板，参数bpColorsTable
 * 指向要替换的伪彩色编码表。
 *
 ************************************************************************/
BOOL WINAPI ReplaceColorPal(LPSTR lpDIB, BYTE * bpColorsTable)
```

```
{
    // 循环变量
    int i;

    // 颜色表中的颜色数目
    WORD wNumColors;

    // 指向BITMAPINFO结构的指针（Win3.0）
    LPBITMAPINFO lpbmi;

    // 指向BITMAPCOREINFO结构的指针
    LPBITMAPCOREINFO lpbmc;

    // 表明是否是Win3.0 DIB的标记
    BOOL bWinStyleDIB;

    // 创建结果
    BOOL bResult = FALSE;

    // 获取指向BITMAPINFO结构的指针（Win3.0）
    lpbmi = (LPBITMAPINFO)lpDIB;

    // 获取指向BITMAPCOREINFO结构的指针
    lpbmc = (LPBITMAPCOREINFO)lpDIB;

    // 获取DIB中颜色表中的颜色数目
    wNumColors = ::DIBNumColors(lpDIB);

    // 判断颜色数目是否是256色
    if (wNumColors == 256)
    {

        // 判断是否是WIN3.0的DIB
        bWinStyleDIB = IS_WIN30_DIB(lpDIB);

        // 读取伪彩色编码，更新DIB调色板
        for (i = 0; i < (int)wNumColors; i++)
        {
            if (bWinStyleDIB)
            {
                // 更新DIB调色板红色分量
                lpbmi->bmiColors[i].rgbRed = bpColorsTable[i * 4];

                // 更新DIB调色板绿色分量
                lpbmi->bmiColors[i].rgbGreen = bpColorsTable[i * 4 + 1];

                // 更新DIB调色板蓝色分量
                lpbmi->bmiColors[i].rgbBlue = bpColorsTable[i * 4 + 2];

                // 更新DIB调色板保留位
```

```
                lpbmi->bmiColors[i].rgbReserved = 0;
            }
            else
            {
                // 更新DIB调色板红色分量
                lpbmc->bmciColors[i].rgbtRed = bpColorsTable[i * 4];

                // 更新DIB调色板绿色分量
                lpbmc->bmciColors[i].rgbtGreen = bpColorsTable[i * 4 + 1];

                // 更新DIB调色板蓝色分量
                lpbmc->bmciColors[i].rgbtBlue = bpColorsTable[i * 4 + 2];
            }
        }
    }

    // 返回
    return bResult;

}
```

下面我们给文档添加一个整型类成员变量 m_nColorIndex 来保存当前使用的伪彩色编码索引，并在菜单伪彩色编码的单击事件中添加如下代码：

```
void CCh1_1View::OnEnhaColor()
{
    // 伪彩色编码

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 保存用户选择的伪彩色编码表索引
    int      nColor;

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（只处理256色位图的伪彩色变换，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的伪彩色变换！", "系统提示",
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
```

```
}

// 参数对话框
CDlgColor dlgPara;

// 初始化变量值
if (pDoc->m_nColorIndex >= 0)
{
    // 初始选中当前的伪彩色
    dlgPara.m_nColor = pDoc->m_nColorIndex;
}
else
{
    // 初始选中灰度伪彩色编码表
    dlgPara.m_nColor = 0;
}

// 指向名称数组的指针
dlgPara.m_lpColorName = (LPSTR) ColorScaleName;

// 伪彩色编码数目
dlgPara.m_nColorCount = COLOR_SCALE_COUNT;

// 名称字符串长度
dlgPara.m_nNameLen = sizeof(ColorScaleName) / COLOR_SCALE_COUNT;

// 显示对话框，提示用户设定平移量
if (dlgPara.DoModal() != IDOK)
{
    // 返回
    return;
}

// 获取用户的设定
nColor = dlgPara.m_nColor;

// 删除对话框
delete dlgPara;

// 更改光标形状
BeginWaitCursor();

// 判断伪彩色编码是否改动
if (pDoc->m_nColorIndex != nColor)
{
    // 调用ReplaceColorPal()函数变换调色板
    ::ReplaceColorPal(lpDIB, (BYTE*) ColorsTable[nColor]);

    // 替换当前文档调色板
    pDoc->GetDocPalette()->SetPaletteEntries(0, 256,
         (LPPALETTEENTRY) ColorsTable[nColor]);
```

```
        // 更新类成员变量
        pDoc->m_nColorIndex = nColor;

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 实现新的调色板
        OnDoRealize((WPARAM)m_hWnd,0);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
}
```

上面代码中用到的常数 COLOR_SCALE_COUNT、字符串数组 ColorScaleName 和伪彩色编码表 ColorsTable 是在伪彩色编码表头文件 ColorTable.h 中定义的，在使用它们之前必须加入下面的# include 语句：

```
#include "ColorTable.h"
```

伪彩色编码表头文件 ColorTable.h 中定义了一些常用的伪彩色编码表，它的完整代码如下：

```
// 编码表个数
#define COLOR_SCALE_COUNT 12

// 编码表名称
const char ColorScaleName[COLOR_SCALE_COUNT][64] =
{
        "  1   常规灰度编码",
        "  2   逆灰度编码",
        "  3   红色饱和度编码",
        "  4   绿色饱和度编码",
        "  5   蓝色饱和度编码",
        "  6   黄色饱和度编码",
        "  7   青色饱和度编码",
        "  8   紫色饱和度编码",
        "  9   彩虹编码 1",
        "  10  彩虹编码 2",
        "  11  热金属编码 1",
        "  12  热金属编码 2",
};

// 编码表RGB数组
const BYTE ColorsTable[COLOR_SCALE_COUNT][256][4] =
```

```
{
{ // 常规灰度编码
{  0,  0,  0,0 }, {  1,  1,  1,0 }, {  2,  2,  2,0 }, {  3,  3,  3,0 }, //4
{  4,  4,  4,0 }, {  5,  5,  5,0 }, {  6,  6,  6,0 }, {  7,  7,  7,0 }, //8
{  8,  8,  8,0 }, {  9,  9,  9,0 }, { 10, 10, 10,0 }, { 11, 11, 11,0 }, //12
{ 12, 12, 12,0 }, { 13, 13, 13,0 }, { 14, 14, 14,0 }, { 15, 15, 15,0 }, //16
{ 16, 16, 16,0 }, { 17, 17, 17,0 }, { 18, 18, 18,0 }, { 19, 19, 19,0 }, //20
{ 20, 20, 20,0 }, { 21, 21, 21,0 }, { 22, 22, 22,0 }, { 23, 23, 23,0 }, //24
{ 24, 24, 24,0 }, { 25, 25, 25,0 }, { 26, 26, 26,0 }, { 27, 27, 27,0 }, //28
{ 28, 28, 28,0 }, { 29, 29, 29,0 }, { 30, 30, 30,0 }, { 31, 31, 31,0 }, //32
{ 32, 32, 32,0 }, { 33, 33, 33,0 }, { 34, 34, 34,0 }, { 35, 35, 35,0 }, //36
{ 36, 36, 36,0 }, { 37, 37, 37,0 }, { 38, 38, 38,0 }, { 39, 39, 39,0 }, //40
{ 40, 40, 40,0 }, { 41, 41, 41,0 }, { 42, 42, 42,0 }, { 43, 43, 43,0 }, //44
{ 44, 44, 44,0 }, { 45, 45, 45,0 }, { 46, 46, 46,0 }, { 47, 47, 47,0 }, //48
{ 48, 48, 48,0 }, { 49, 49, 49,0 }, { 50, 50, 50,0 }, { 51, 51, 51,0 }, //52
{ 52, 52, 52,0 }, { 53, 53, 53,0 }, { 54, 54, 54,0 }, { 55, 55, 55,0 }, //56
{ 56, 56, 56,0 }, { 57, 57, 57,0 }, { 58, 58, 58,0 }, { 59, 59, 59,0 }, //60
{ 60, 60, 60,0 }, { 61, 61, 61,0 }, { 62, 62, 62,0 }, { 63, 63, 63,0 }, //64
{ 64, 64, 64,0 }, { 65, 65, 65,0 }, { 66, 66, 66,0 }, { 67, 67, 67,0 }, //68
{ 68, 68, 68,0 }, { 69, 69, 69,0 }, { 70, 70, 70,0 }, { 71, 71, 71,0 }, //72
{ 72, 72, 72,0 }, { 73, 73, 73,0 }, { 74, 74, 74,0 }, { 75, 75, 75,0 }, //76
{ 76, 76, 76,0 }, { 77, 77, 77,0 }, { 78, 78, 78,0 }, { 79, 79, 79,0 }, //80
{ 80, 80, 80,0 }, { 81, 81, 81,0 }, { 82, 82, 82,0 }, { 83, 83, 83,0 }, //84
{ 84, 84, 84,0 }, { 85, 85, 85,0 }, { 86, 86, 86,0 }, { 87, 87, 87,0 }, //88
{ 88, 88, 88,0 }, { 89, 89, 89,0 }, { 90, 90, 90,0 }, { 91, 91, 91,0 }, //92
{ 92, 92, 92,0 }, { 93, 93, 93,0 }, { 94, 94, 94,0 }, { 95, 95, 95,0 }, //96
{ 96, 96, 96,0 }, { 97, 97, 97,0 }, { 98, 98, 98,0 }, { 99, 99, 99,0 }, //100
{ 100,100,100,0 }, { 101,101,101,0 }, { 102,102,102,0 }, { 103,103,103,0 }, //104
{ 104,104,104,0 }, { 105,105,105,0 }, { 106,106,106,0 }, { 107,107,107,0 }, //108
{ 108,108,108,0 }, { 109,109,109,0 }, { 110,110,110,0 }, { 111,111,111,0 }, //112
{ 112,112,112,0 }, { 113,113,113,0 }, { 114,114,114,0 }, { 115,115,115,0 }, //116
{ 116,116,116,0 }, { 117,117,117,0 }, { 118,118,118,0 }, { 119,119,119,0 }, //120
{ 120,120,120,0 }, { 121,121,121,0 }, { 122,122,122,0 }, { 123,123,123,0 }, //124
{ 124,124,124,0 }, { 125,125,125,0 }, { 126,126,126,0 }, { 127,127,127,0 }, //128
{ 128,128,128,0 }, { 129,129,129,0 }, { 130,130,130,0 }, { 131,131,131,0 }, //132
{ 132,132,132,0 }, { 133,133,133,0 }, { 134,134,134,0 }, { 135,135,135,0 }, //136
{ 136,136,136,0 }, { 137,137,137,0 }, { 138,138,138,0 }, { 139,139,139,0 }, //140
{ 140,140,140,0 }, { 141,141,141,0 }, { 142,142,142,0 }, { 143,143,143,0 }, //144
{ 144,144,144,0 }, { 145,145,145,0 }, { 146,146,146,0 }, { 147,147,147,0 }, //148
{ 148,148,148,0 }, { 149,149,149,0 }, { 150,150,150,0 }, { 151,151,151,0 }, //152
{ 152,152,152,0 }, { 153,153,153,0 }, { 154,154,154,0 }, { 155,155,155,0 }, //156
{ 156,156,156,0 }, { 157,157,157,0 }, { 158,158,158,0 }, { 159,159,159,0 }, //160
{ 160,160,160,0 }, { 161,161,161,0 }, { 162,162,162,0 }, { 163,163,163,0 }, //164
{ 164,164,164,0 }, { 165,165,165,0 }, { 166,166,166,0 }, { 167,167,167,0 }, //168
{ 168,168,168,0 }, { 169,169,169,0 }, { 170,170,170,0 }, { 171,171,171,0 }, //172
{ 172,172,172,0 }, { 173,173,173,0 }, { 174,174,174,0 }, { 175,175,175,0 }, //176
{ 176,176,176,0 }, { 177,177,177,0 }, { 178,178,178,0 }, { 179,179,179,0 }, //180
{ 180,180,180,0 }, { 181,181,181,0 }, { 182,182,182,0 }, { 183,183,183,0 }, //184
{ 184,184,184,0 }, { 185,185,185,0 }, { 186,186,186,0 }, { 187,187,187,0 }, //188
{ 188,188,188,0 }, { 189,189,189,0 }, { 190,190,190,0 }, { 191,191,191,0 }, //192
{ 192,192,192,0 }, { 193,193,193,0 }, { 194,194,194,0 }, { 195,195,195,0 }, //196
```

```
{ 196, 196, 196, 0 }, { 197, 197, 197, 0 }, { 198, 198, 198, 0 }, { 199, 199, 199, 0 }, //200
{ 200, 200, 200, 0 }, { 201, 201, 201, 0 }, { 202, 202, 202, 0 }, { 203, 203, 203, 0 }, //204
{ 204, 204, 204, 0 }, { 205, 205, 205, 0 }, { 206, 206, 206, 0 }, { 207, 207, 207, 0 }, //208
{ 208, 208, 208, 0 }, { 209, 209, 209, 0 }, { 210, 210, 210, 0 }, { 211, 211, 211, 0 }, //212
{ 212, 212, 212, 0 }, { 213, 213, 213, 0 }, { 214, 214, 214, 0 }, { 215, 215, 215, 0 }, //216
{ 216, 216, 216, 0 }, { 217, 217, 217, 0 }, { 218, 218, 218, 0 }, { 219, 219, 219, 0 }, //220
{ 220, 220, 220, 0 }, { 221, 221, 221, 0 }, { 222, 222, 222, 0 }, { 223, 223, 223, 0 }, //224
{ 224, 224, 224, 0 }, { 225, 225, 225, 0 }, { 226, 226, 226, 0 }, { 227, 227, 227, 0 }, //228
{ 228, 228, 228, 0 }, { 229, 229, 229, 0 }, { 230, 230, 230, 0 }, { 231, 231, 231, 0 }, //232
{ 232, 232, 232, 0 }, { 233, 233, 233, 0 }, { 234, 234, 234, 0 }, { 235, 235, 235, 0 }, //236
{ 236, 236, 236, 0 }, { 237, 237, 237, 0 }, { 238, 238, 238, 0 }, { 239, 239, 239, 0 }, //240
{ 240, 240, 240, 0 }, { 241, 241, 241, 0 }, { 242, 242, 242, 0 }, { 243, 243, 243, 0 }, //244
{ 244, 244, 244, 0 }, { 245, 245, 245, 0 }, { 246, 246, 246, 0 }, { 247, 247, 247, 0 }, //248
{ 248, 248, 248, 0 }, { 249, 249, 249, 0 }, { 250, 250, 250, 0 }, { 251, 251, 251, 0 }, //252
{ 252, 252, 252, 0 }, { 253, 253, 253, 0 }, { 254, 254, 254, 0 }, { 255, 255, 255, 0 }, //256

// 逆灰度编码
{ 255, 255, 255, 0 }, { 254, 254, 254, 0 }, { 253, 253, 253, 0 }, { 252, 252, 252, 0 }, //4
{ 251, 251, 251, 0 }, { 250, 250, 250, 0 }, { 249, 249, 249, 0 }, { 248, 248, 248, 0 }, //8
{ 247, 247, 247, 0 }, { 246, 246, 246, 0 }, { 245, 245, 245, 0 }, { 244, 244, 244, 0 }, //12
{ 243, 243, 243, 0 }, { 242, 242, 242, 0 }, { 241, 241, 241, 0 }, { 240, 240, 240, 0 }, //16
{ 239, 239, 239, 0 }, { 238, 238, 238, 0 }, { 237, 237, 237, 0 }, { 236, 236, 236, 0 }, //20
{ 235, 235, 235, 0 }, { 234, 234, 234, 0 }, { 233, 233, 233, 0 }, { 232, 232, 232, 0 }, //24
{ 231, 231, 231, 0 }, { 230, 230, 230, 0 }, { 229, 229, 229, 0 }, { 228, 228, 228, 0 }, //28
{ 227, 227, 227, 0 }, { 226, 226, 226, 0 }, { 225, 225, 225, 0 }, { 224, 224, 224, 0 }, //32
{ 223, 223, 223, 0 }, { 222, 222, 222, 0 }, { 221, 221, 221, 0 }, { 220, 220, 220, 0 }, //36
{ 219, 219, 219, 0 }, { 218, 218, 218, 0 }, { 217, 217, 217, 0 }, { 216, 216, 216, 0 }, //40
{ 215, 215, 215, 0 }, { 214, 214, 214, 0 }, { 213, 213, 213, 0 }, { 212, 212, 212, 0 }, //44
{ 211, 211, 211, 0 }, { 210, 210, 210, 0 }, { 209, 209, 209, 0 }, { 208, 208, 208, 0 }, //48
{ 207, 207, 207, 0 }, { 206, 206, 206, 0 }, { 205, 205, 205, 0 }, { 204, 204, 204, 0 }, //52
{ 203, 203, 203, 0 }, { 202, 202, 202, 0 }, { 201, 201, 201, 0 }, { 200, 200, 200, 0 }, //56
{ 199, 199, 199, 0 }, { 198, 198, 198, 0 }, { 197, 197, 197, 0 }, { 196, 196, 196, 0 }, //60
{ 195, 195, 195, 0 }, { 194, 194, 194, 0 }, { 193, 193, 193, 0 }, { 192, 192, 192, 0 }, //64
{ 191, 191, 191, 0 }, { 190, 190, 190, 0 }, { 189, 189, 189, 0 }, { 188, 188, 188, 0 }, //68
{ 187, 187, 187, 0 }, { 186, 186, 186, 0 }, { 185, 185, 185, 0 }, { 184, 184, 184, 0 }, //72
{ 183, 183, 183, 0 }, { 182, 182, 182, 0 }, { 181, 181, 181, 0 }, { 180, 180, 180, 0 }, //76
{ 179, 179, 179, 0 }, { 178, 178, 178, 0 }, { 177, 177, 177, 0 }, { 176, 176, 176, 0 }, //80
{ 175, 175, 175, 0 }, { 174, 174, 174, 0 }, { 173, 173, 173, 0 }, { 172, 172, 172, 0 }, //84
{ 171, 171, 171, 0 }, { 170, 170, 170, 0 }, { 169, 169, 169, 0 }, { 168, 168, 168, 0 }, //88
{ 167, 167, 167, 0 }, { 166, 166, 166, 0 }, { 165, 165, 165, 0 }, { 164, 164, 164, 0 }, //92
{ 163, 163, 163, 0 }, { 162, 162, 162, 0 }, { 161, 161, 161, 0 }, { 160, 160, 160, 0 }, //96
{ 159, 159, 159, 0 }, { 158, 158, 158, 0 }, { 157, 157, 157, 0 }, { 156, 156, 156, 0 }, //100
{ 155, 155, 155, 0 }, { 154, 154, 154, 0 }, { 153, 153, 153, 0 }, { 152, 152, 152, 0 }, //104
{ 151, 151, 151, 0 }, { 150, 150, 150, 0 }, { 149, 149, 149, 0 }, { 148, 148, 148, 0 }, //108
{ 147, 147, 147, 0 }, { 146, 146, 146, 0 }, { 145, 145, 145, 0 }, { 144, 144, 144, 0 }, //112
{ 143, 143, 143, 0 }, { 142, 142, 142, 0 }, { 141, 141, 141, 0 }, { 140, 140, 140, 0 }, //116
{ 139, 139, 139, 0 }, { 138, 138, 138, 0 }, { 137, 137, 137, 0 }, { 136, 136, 136, 0 }, //120
{ 135, 135, 135, 0 }, { 134, 134, 134, 0 }, { 133, 133, 133, 0 }, { 132, 132, 132, 0 }, //124
{ 131, 131, 131, 0 }, { 130, 130, 130, 0 }, { 129, 129, 129, 0 }, { 128, 128, 128, 0 }, //128
{ 127, 127, 127, 0 }, { 126, 126, 126, 0 }, { 125, 125, 125, 0 }, { 124, 124, 124, 0 }, //132
{ 123, 123, 123, 0 }, { 122, 122, 122, 0 }, { 121, 121, 121, 0 }, { 120, 120, 120, 0 }, //136
```

```
    { 119,119,119,0 }, { 118,118,118,0 }, { 117,117,117,0 }, { 116,116,116,0 }, //140
    { 115,115,115,0 }, { 114,114,114,0 }, { 113,113,113,0 }, { 112,112,112,0 }, //144
    { 111,111,111,0 }, { 110,110,110,0 }, { 109,109,109,0 }, { 108,108,108,0 }, //148
    { 107,107,107,0 }, { 106,106,106,0 }, { 105,105,105,0 }, { 104,104,104,0 }, //152
    { 103,103,103,0 }, { 102,102,102,0 }, { 101,101,101,0 }, { 100,100,100,0 }, //156
    {  99, 99, 99,0 }, {  98, 98, 98,0 }, {  97, 97, 97,0 }, {  96, 96, 96,0 }, //160
    {  95, 95, 95,0 }, {  94, 94, 94,0 }, {  93, 93, 93,0 }, {  92, 92, 92,0 }, //164
    {  91, 91, 91,0 }, {  90, 90, 90,0 }, {  89, 89, 89,0 }, {  88, 88, 88,0 }, //168
    {  87, 87, 87,0 }, {  86, 86, 86,0 }, {  85, 85, 85,0 }, {  84, 84, 84,0 }, //172
    {  83, 83, 83,0 }, {  82, 82, 82,0 }, {  81, 81, 81,0 }, {  80, 80, 80,0 }, //176
    {  79, 79, 79,0 }, {  78, 78, 78,0 }, {  77, 77, 77,0 }, {  76, 76, 76,0 }, //180
    {  75, 75, 75,0 }, {  74, 74, 74,0 }, {  73, 73, 73,0 }, {  72, 72, 72,0 }, //184
    {  71, 71, 71,0 }, {  70, 70, 70,0 }, {  69, 69, 69,0 }, {  68, 68, 68,0 }, //188
    {  67, 67, 67,0 }, {  66, 66, 66,0 }, {  65, 65, 65,0 }, {  64, 64, 64,0 }, //192
    {  63, 63, 63,0 }, {  62, 62, 62,0 }, {  61, 61, 61,0 }, {  60, 60, 60,0 }, //196
    {  59, 59, 59,0 }, {  58, 58, 58,0 }, {  57, 57, 57,0 }, {  56, 56, 56,0 }, //200
    {  55, 55, 55,0 }, {  54, 54, 54,0 }, {  53, 53, 53,0 }, {  52, 52, 52,0 }, //204
    {  51, 51, 51,0 }, {  50, 50, 50,0 }, {  49, 49, 49,0 }, {  48, 48, 48,0 }, //208
    {  47, 47, 47,0 }, {  46, 46, 46,0 }, {  45, 45, 45,0 }, {  44, 44, 44,0 }, //212
    {  43, 43, 43,0 }, {  42, 42, 42,0 }, {  41, 41, 41,0 }, {  40, 40, 40,0 }, //216
    {  39, 39, 39,0 }, {  38, 38, 38,0 }, {  37, 37, 37,0 }, {  36, 36, 36,0 }, //220
    {  35, 35, 35,0 }, {  34, 34, 34,0 }, {  33, 33, 33,0 }, {  32, 32, 32,0 }, //224
    {  31, 31, 31,0 }, {  30, 30, 30,0 }, {  29, 29, 29,0 }, {  28, 28, 28,0 }, //228
    {  27, 27, 27,0 }, {  26, 26, 26,0 }, {  25, 25, 25,0 }, {  24, 24, 24,0 }, //232
    {  23, 23, 23,0 }, {  22, 22, 22,0 }, {  21, 21, 21,0 }, {  20, 20, 20,0 }, //236
    {  19, 19, 19,0 }, {  18, 18, 18,0 }, {  17, 17, 17,0 }, {  16, 16, 16,0 }, //240
    {  15, 15, 15,0 }, {  14, 14, 14,0 }, {  13, 13, 13,0 }, {  12, 12, 12,0 }, //244
    {  11, 11, 11,0 }, {  10, 10, 10,0 }, {   9,  9,  9,0 }, {   8,  8,  8,0 }, //248
    {   7,  7,  7,0 }, {   6,  6,  6,0 }, {   5,  5,  5,0 }, {   4,  4,  4,0 }, //252
    {   3,  3,  3,0 }, {   2,  2,  2,0 }, {   1,  1,  1,0 }, {   0,  0,  0,0 }, //256
},
{   // 红色饱和度编码
    {   0,  0,  0,0 }, {   1,  0,  0,0 }, {   2,  0,  0,0 }, {   3,  0,  0,0 }, //4
    {   4,  0,  0,0 }, {   5,  0,  0,0 }, {   6,  0,  0,0 }, {   7,  0,  0,0 }, //8
    {   8,  0,  0,0 }, {   9,  0,  0,0 }, {  10,  0,  0,0 }, {  11,  0,  0,0 }, //12
    {  12,  0,  0,0 }, {  13,  0,  0,0 }, {  14,  0,  0,0 }, {  15,  0,  0,0 }, //16
    {  16,  0,  0,0 }, {  17,  0,  0,0 }, {  18,  0,  0,0 }, {  19,  0,  0,0 }, //20
    {  20,  0,  0,0 }, {  21,  0,  0,0 }, {  22,  0,  0,0 }, {  23,  0,  0,0 }, //24
    {  24,  0,  0,0 }, {  25,  0,  0,0 }, {  26,  0,  0,0 }, {  27,  0,  0,0 }, //28
    {  28,  0,  0,0 }, {  29,  0,  0,0 }, {  30,  0,  0,0 }, {  31,  0,  0,0 }, //32
    {  32,  0,  0,0 }, {  33,  0,  0,0 }, {  34,  0,  0,0 }, {  35,  0,  0,0 }, //36
    {  36,  0,  0,0 }, {  37,  0,  0,0 }, {  38,  0,  0,0 }, {  39,  0,  0,0 }, //40
    {  40,  0,  0,0 }, {  41,  0,  0,0 }, {  42,  0,  0,0 }, {  43,  0,  0,0 }, //44
    {  44,  0,  0,0 }, {  45,  0,  0,0 }, {  46,  0,  0,0 }, {  47,  0,  0,0 }, //48
    {  48,  0,  0,0 }, {  49,  0,  0,0 }, {  50,  0,  0,0 }, {  51,  0,  0,0 }, //52
    {  52,  0,  0,0 }, {  53,  0,  0,0 }, {  54,  0,  0,0 }, {  55,  0,  0,0 }, //56
    {  56,  0,  0,0 }, {  57,  0,  0,0 }, {  58,  0,  0,0 }, {  59,  0,  0,0 }, //60
    {  60,  0,  0,0 }, {  61,  0,  0,0 }, {  62,  0,  0,0 }, {  63,  0,  0,0 }, //64
    {  64,  0,  0,0 }, {  65,  0,  0,0 }, {  66,  0,  0,0 }, {  67,  0,  0,0 }, //68
    {  68,  0,  0,0 }, {  69,  0,  0,0 }, {  70,  0,  0,0 }, {  71,  0,  0,0 }, //72
    {  72,  0,  0,0 }, {  73,  0,  0,0 }, {  74,  0,  0,0 }, {  75,  0,  0,0 }, //76
```

```
{ 76,  0,  0,0 }, { 77,  0,  0,0 }, { 78,  0,  0,0 }, { 79,  0,  0,0 }, //80
{ 80,  0,  0,0 }, { 81,  0,  0,0 }, { 82,  0,  0,0 }, { 83,  0,  0,0 }, //84
{ 84,  0,  0,0 }, { 85,  0,  0,0 }, { 86,  0,  0,0 }, { 87,  0,  0,0 }, //88
{ 88,  0,  0,0 }, { 89,  0,  0,0 }, { 90,  0,  0,0 }, { 91,  0,  0,0 }, //92
{ 92,  0,  0,0 }, { 93,  0,  0,0 }, { 94,  0,  0,0 }, { 95,  0,  0,0 }, //96
{ 96,  0,  0,0 }, { 97,  0,  0,0 }, { 98,  0,  0,0 }, { 99,  0,  0,0 }, //100
{ 100, 0,  0,0 }, { 101, 0,  0,0 }, { 102, 0,  0,0 }, { 103, 0,  0,0 }, //104
{ 104, 0,  0,0 }, { 105, 0,  0,0 }, { 106, 0,  0,0 }, { 107, 0,  0,0 }, //108
{ 108, 0,  0,0 }, { 109, 0,  0,0 }, { 110, 0,  0,0 }, { 111, 0,  0,0 }, //112
{ 112, 0,  0,0 }, { 113, 0,  0,0 }, { 114, 0,  0,0 }, { 115, 0,  0,0 }, //116
{ 116, 0,  0,0 }, { 117, 0,  0,0 }, { 118, 0,  0,0 }, { 119, 0,  0,0 }, //120
{ 120, 0,  0,0 }, { 121, 0,  0,0 }, { 122, 0,  0,0 }, { 123, 0,  0,0 }, //124
{ 124, 0,  0,0 }, { 125, 0,  0,0 }, { 126, 0,  0,0 }, { 127, 0,  0,0 }, //128
{ 128, 0,  0,0 }, { 129, 0,  0,0 }, { 130, 0,  0,0 }, { 131, 0,  0,0 }, //132
{ 132, 0,  0,0 }, { 133, 0,  0,0 }, { 134, 0,  0,0 }, { 135, 0,  0,0 }, //136
{ 136, 0,  0,0 }, { 137, 0,  0,0 }, { 138, 0,  0,0 }, { 139, 0,  0,0 }, //140
{ 140, 0,  0,0 }, { 141, 0,  0,0 }, { 142, 0,  0,0 }, { 143, 0,  0,0 }, //144
{ 144, 0,  0,0 }, { 145, 0,  0,0 }, { 146, 0,  0,0 }, { 147, 0,  0,0 }, //148
{ 148, 0,  0,0 }, { 149, 0,  0,0 }, { 150, 0,  0,0 }, { 151, 0,  0,0 }, //152
{ 152, 0,  0,0 }, { 153, 0,  0,0 }, { 154, 0,  0,0 }, { 155, 0,  0,0 }, //156
{ 156, 0,  0,0 }, { 157, 0,  0,0 }, { 158, 0,  0,0 }, { 159, 0,  0,0 }, //160
{ 160, 0,  0,0 }, { 161, 0,  0,0 }, { 162, 0,  0,0 }, { 163, 0,  0,0 }, //164
{ 164, 0,  0,0 }, { 165, 0,  0,0 }, { 166, 0,  0,0 }, { 167, 0,  0,0 }, //168
{ 168, 0,  0,0 }, { 169, 0,  0,0 }, { 170, 0,  0,0 }, { 171, 0,  0,0 }, //172
{ 172, 0,  0,0 }, { 173, 0,  0,0 }, { 174, 0,  0,0 }, { 175, 0,  0,0 }, //176
{ 176, 0,  0,0 }, { 177, 0,  0,0 }, { 178, 0,  0,0 }, { 179, 0,  0,0 }, //180
{ 180, 0,  0,0 }, { 181, 0,  0,0 }, { 182, 0,  0,0 }, { 183, 0,  0,0 }, //184
{ 184, 0,  0,0 }, { 185, 0,  0,0 }, { 186, 0,  0,0 }, { 187, 0,  0,0 }, //188
{ 188, 0,  0,0 }, { 189, 0,  0,0 }, { 190, 0,  0,0 }, { 191, 0,  0,0 }, //192
{ 192, 0,  0,0 }, { 193, 0,  0,0 }, { 194, 0,  0,0 }, { 195, 0,  0,0 }, //196
{ 196, 0,  0,0 }, { 197, 0,  0,0 }, { 198, 0,  0,0 }, { 199, 0,  0,0 }, //200
{ 200, 0,  0,0 }, { 201, 0,  0,0 }, { 202, 0,  0,0 }, { 203, 0,  0,0 }, //204
{ 204, 0,  0,0 }, { 205, 0,  0,0 }, { 206, 0,  0,0 }, { 207, 0,  0,0 }, //208
{ 208, 0,  0,0 }, { 209, 0,  0,0 }, { 210, 0,  0,0 }, { 211, 0,  0,0 }, //212
{ 212, 0,  0,0 }, { 213, 0,  0,0 }, { 214, 0,  0,0 }, { 215, 0,  0,0 }, //216
{ 216, 0,  0,0 }, { 217, 0,  0,0 }, { 218, 0,  0,0 }, { 219, 0,  0,0 }, //220
{ 220, 0,  0,0 }, { 221, 0,  0,0 }, { 222, 0,  0,0 }, { 223, 0,  0,0 }, //224
{ 224, 0,  0,0 }, { 225, 0,  0,0 }, { 226, 0,  0,0 }, { 227, 0,  0,0 }, //228
{ 228, 0,  0,0 }, { 229, 0,  0,0 }, { 230, 0,  0,0 }, { 231, 0,  0,0 }, //232
{ 232, 0,  0,0 }, { 233, 0,  0,0 }, { 234, 0,  0,0 }, { 235, 0,  0,0 }, //236
{ 236, 0,  0,0 }, { 237, 0,  0,0 }, { 238, 0,  0,0 }, { 239, 0,  0,0 }, //240
{ 240, 0,  0,0 }, { 241, 0,  0,0 }, { 242, 0,  0,0 }, { 243, 0,  0,0 }, //244
{ 244, 0,  0,0 }, { 245, 0,  0,0 }, { 246, 0,  0,0 }, { 247, 0,  0,0 }, //248
{ 248, 0,  0,0 }, { 249, 0,  0,0 }, { 250, 0,  0,0 }, { 251, 0,  0,0 }, //252
{ 252, 0,  0,0 }, { 253, 0,  0,0 }, { 254, 0,  0,0 }, { 255, 0,  0,0 }, //256
},
{   // 绿色饱和度编码
{ 0,  0,  0,0 }, { 0,  1,  0,0 }, { 0,  2,  0,0 }, { 0,  3,  0,0 }, //4
{ 0,  4,  0,0 }, { 0,  5,  0,0 }, { 0,  6,  0,0 }, { 0,  7,  0,0 }, //8
{ 0,  8,  0,0 }, { 0,  9,  0,0 }, { 0, 10,  0,0 }, { 0, 11,  0,0 }, //12
{ 0, 12,  0,0 }, { 0, 13,  0,0 }, { 0, 14,  0,0 }, { 0, 15,  0,0 }, //16
```

```
{  0,  16,   0,0 },  {  0,  17,   0,0 },  {  0,  18,   0,0 },  {  0,  19,   0,0 },  //20
{  0,  20,   0,0 },  {  0,  21,   0,0 },  {  0,  22,   0,0 },  {  0,  23,   0,0 },  //24
{  0,  24,   0,0 },  {  0,  25,   0,0 },  {  0,  26,   0,0 },  {  0,  27,   0,0 },  //28
{  0,  28,   0,0 },  {  0,  29,   0,0 },  {  0,  30,   0,0 },  {  0,  31,   0,0 },  //32
{  0,  32,   0,0 },  {  0,  33,   0,0 },  {  0,  34,   0,0 },  {  0,  35,   0,0 },  //36
{  0,  36,   0,0 },  {  0,  37,   0,0 },  {  0,  38,   0,0 },  {  0,  39,   0,0 },  //40
{  0,  40,   0,0 },  {  0,  41,   0,0 },  {  0,  42,   0,0 },  {  0,  43,   0,0 },  //44
{  0,  44,   0,0 },  {  0,  45,   0,0 },  {  0,  46,   0,0 },  {  0,  47,   0,0 },  //48
{  0,  48,   0,0 },  {  0,  49,   0,0 },  {  0,  50,   0,0 },  {  0,  51,   0,0 },  //52
{  0,  52,   0,0 },  {  0,  53,   0,0 },  {  0,  54,   0,0 },  {  0,  55,   0,0 },  //56
{  0,  56,   0,0 },  {  0,  57,   0,0 },  {  0,  58,   0,0 },  {  0,  59,   0,0 },  //60
{  0,  60,   0,0 },  {  0,  61,   0,0 },  {  0,  62,   0,0 },  {  0,  63,   0,0 },  //64
{  0,  64,   0,0 },  {  0,  65,   0,0 },  {  0,  66,   0,0 },  {  0,  67,   0,0 },  //68
{  0,  68,   0,0 },  {  0,  69,   0,0 },  {  0,  70,   0,0 },  {  0,  71,   0,0 },  //72
{  0,  72,   0,0 },  {  0,  73,   0,0 },  {  0,  74,   0,0 },  {  0,  75,   0,0 },  //76
{  0,  76,   0,0 },  {  0,  77,   0,0 },  {  0,  78,   0,0 },  {  0,  79,   0,0 },  //80
{  0,  80,   0,0 },  {  0,  81,   0,0 },  {  0,  82,   0,0 },  {  0,  83,   0,0 },  //84
{  0,  84,   0,0 },  {  0,  85,   0,0 },  {  0,  86,   0,0 },  {  0,  87,   0,0 },  //88
{  0,  88,   0,0 },  {  0,  89,   0,0 },  {  0,  90,   0,0 },  {  0,  91,   0,0 },  //92
{  0,  92,   0,0 },  {  0,  93,   0,0 },  {  0,  94,   0,0 },  {  0,  95,   0,0 },  //96
{  0,  96,   0,0 },  {  0,  97,   0,0 },  {  0,  98,   0,0 },  {  0,  99,   0,0 },  //100
{  0, 100,   0,0 },  {  0, 101,   0,0 },  {  0, 102,   0,0 },  {  0, 103,   0,0 },  //104
{  0, 104,   0,0 },  {  0, 105,   0,0 },  {  0, 106,   0,0 },  {  0, 107,   0,0 },  //108
{  0, 108,   0,0 },  {  0, 109,   0,0 },  {  0, 110,   0,0 },  {  0, 111,   0,0 },  //112
{  0, 112,   0,0 },  {  0, 113,   0,0 },  {  0, 114,   0,0 },  {  0, 115,   0,0 },  //116
{  0, 116,   0,0 },  {  0, 117,   0,0 },  {  0, 118,   0,0 },  {  0, 119,   0,0 },  //120
{  0, 120,   0,0 },  {  0, 121,   0,0 },  {  0, 122,   0,0 },  {  0, 123,   0,0 },  //124
{  0, 124,   0,0 },  {  0, 125,   0,0 },  {  0, 126,   0,0 },  {  0, 127,   0,0 },  //128
{  0, 128,   0,0 },  {  0, 129,   0,0 },  {  0, 130,   0,0 },  {  0, 131,   0,0 },  //132
{  0, 132,   0,0 },  {  0, 133,   0,0 },  {  0, 134,   0,0 },  {  0, 135,   0,0 },  //136
{  0, 136,   0,0 },  {  0, 137,   0,0 },  {  0, 138,   0,0 },  {  0, 139,   0,0 },  //140
{  0, 140,   0,0 },  {  0, 141,   0,0 },  {  0, 142,   0,0 },  {  0, 143,   0,0 },  //144
{  0, 144,   0,0 },  {  0, 145,   0,0 },  {  0, 146,   0,0 },  {  0, 147,   0,0 },  //148
{  0, 148,   0,0 },  {  0, 149,   0,0 },  {  0, 150,   0,0 },  {  0, 151,   0,0 },  //152
{  0, 152,   0,0 },  {  0, 153,   0,0 },  {  0, 154,   0,0 },  {  0, 155,   0,0 },  //156
{  0, 156,   0,0 },  {  0, 157,   0,0 },  {  0, 158,   0,0 },  {  0, 159,   0,0 },  //160
{  0, 160,   0,0 },  {  0, 161,   0,0 },  {  0, 162,   0,0 },  {  0, 163,   0,0 },  //164
{  0, 164,   0,0 },  {  0, 165,   0,0 },  {  0, 166,   0,0 },  {  0, 167,   0,0 },  //168
{  0, 168,   0,0 },  {  0, 169,   0,0 },  {  0, 170,   0,0 },  {  0, 171,   0,0 },  //172
{  0, 172,   0,0 },  {  0, 173,   0,0 },  {  0, 174,   0,0 },  {  0, 175,   0,0 },  //176
{  0, 176,   0,0 },  {  0, 177,   0,0 },  {  0, 178,   0,0 },  {  0, 179,   0,0 },  //180
{  0, 180,   0,0 },  {  0, 181,   0,0 },  {  0, 182,   0,0 },  {  0, 183,   0,0 },  //184
{  0, 184,   0,0 },  {  0, 185,   0,0 },  {  0, 186,   0,0 },  {  0, 187,   0,0 },  //188
{  0, 188,   0,0 },  {  0, 189,   0,0 },  {  0, 190,   0,0 },  {  0, 191,   0,0 },  //192
{  0, 192,   0,0 },  {  0, 193,   0,0 },  {  0, 194,   0,0 },  {  0, 195,   0,0 },  //196
{  0, 196,   0,0 },  {  0, 197,   0,0 },  {  0, 198,   0,0 },  {  0, 199,   0,0 },  //200
{  0, 200,   0,0 },  {  0, 201,   0,0 },  {  0, 202,   0,0 },  {  0, 203,   0,0 },  //204
{  0, 204,   0,0 },  {  0, 205,   0,0 },  {  0, 206,   0,0 },  {  0, 207,   0,0 },  //208
{  0, 208,   0,0 },  {  0, 209,   0,0 },  {  0, 210,   0,0 },  {  0, 211,   0,0 },  //212
{  0, 212,   0,0 },  {  0, 213,   0,0 },  {  0, 214,   0,0 },  {  0, 215,   0,0 },  //216
{  0, 216,   0,0 },  {  0, 217,   0,0 },  {  0, 218,   0,0 },  {  0, 219,   0,0 },  //220
```

```
{ 0,220, 0,0 }, { 0,221, 0,0 }, { 0,222, 0,0 }, { 0,223, 0,0 }, //224
{ 0,224, 0,0 }, { 0,225, 0,0 }, { 0,226, 0,0 }, { 0,227, 0,0 }, //228
{ 0,228, 0,0 }, { 0,229, 0,0 }, { 0,230, 0,0 }, { 0,231, 0,0 }, //232
{ 0,232, 0,0 }, { 0,233, 0,0 }, { 0,234, 0,0 }, { 0,235, 0,0 }, //236
{ 0,236, 0,0 }, { 0,237, 0,0 }, { 0,238, 0,0 }, { 0,239, 0,0 }, //240
{ 0,240, 0,0 }, { 0,241, 0,0 }, { 0,242, 0,0 }, { 0,243, 0,0 }, //244
{ 0,244, 0,0 }, { 0,245, 0,0 }, { 0,246, 0,0 }, { 0,247, 0,0 }, //248
{ 0,248, 0,0 }, { 0,249, 0,0 }, { 0,250, 0,0 }, { 0,251, 0,0 }, //252
{ 0,252, 0,0 }, { 0,253, 0,0 }, { 0,254, 0,0 }, { 0,255, 0,0 }, //256
},
{ // 蓝色饱和度编码
{ 0, 0, 0,0 }, { 0, 0, 1,0 }, { 0, 0, 2,0 }, { 0, 0, 3,0 }, //4
{ 0, 0, 4,0 }, { 0, 0, 5,0 }, { 0, 0, 6,0 }, { 0, 0, 7,0 }, //8
{ 0, 0, 8,0 }, { 0, 0, 9,0 }, { 0, 0, 10,0 }, { 0, 0, 11,0 }, //12
{ 0, 0, 12,0 }, { 0, 0, 13,0 }, { 0, 0, 14,0 }, { 0, 0, 15,0 }, //16
{ 0, 0, 16,0 }, { 0, 0, 17,0 }, { 0, 0, 18,0 }, { 0, 0, 19,0 }, //20
{ 0, 0, 20,0 }, { 0, 0, 21,0 }, { 0, 0, 22,0 }, { 0, 0, 23,0 }, //24
{ 0, 0, 24,0 }, { 0, 0, 25,0 }, { 0, 0, 26,0 }, { 0, 0, 27,0 }, //28
{ 0, 0, 28,0 }, { 0, 0, 29,0 }, { 0, 0, 30,0 }, { 0, 0, 31,0 }, //32
{ 0, 0, 32,0 }, { 0, 0, 33,0 }, { 0, 0, 34,0 }, { 0, 0, 35,0 }, //36
{ 0, 0, 36,0 }, { 0, 0, 37,0 }, { 0, 0, 38,0 }, { 0, 0, 39,0 }, //40
{ 0, 0, 40,0 }, { 0, 0, 41,0 }, { 0, 0, 42,0 }, { 0, 0, 43,0 }, //44
{ 0, 0, 44,0 }, { 0, 0, 45,0 }, { 0, 0, 46,0 }, { 0, 0, 47,0 }, //48
{ 0, 0, 48,0 }, { 0, 0, 49,0 }, { 0, 0, 50,0 }, { 0, 0, 51,0 }, //52
{ 0, 0, 52,0 }, { 0, 0, 53,0 }, { 0, 0, 54,0 }, { 0, 0, 55,0 }, //56
{ 0, 0, 56,0 }, { 0, 0, 57,0 }, { 0, 0, 58,0 }, { 0, 0, 59,0 }, //60
{ 0, 0, 60,0 }, { 0, 0, 61,0 }, { 0, 0, 62,0 }, { 0, 0, 63,0 }, //64
{ 0, 0, 64,0 }, { 0, 0, 65,0 }, { 0, 0, 66,0 }, { 0, 0, 67,0 }, //68
{ 0, 0, 68,0 }, { 0, 0, 69,0 }, { 0, 0, 70,0 }, { 0, 0, 71,0 }, //72
{ 0, 0, 72,0 }, { 0, 0, 73,0 }, { 0, 0, 74,0 }, { 0, 0, 75,0 }, //76
{ 0, 0, 76,0 }, { 0, 0, 77,0 }, { 0, 0, 78,0 }, { 0, 0, 79,0 }, //80
{ 0, 0, 80,0 }, { 0, 0, 81,0 }, { 0, 0, 82,0 }, { 0, 0, 83,0 }, //84
{ 0, 0, 84,0 }, { 0, 0, 85,0 }, { 0, 0, 86,0 }, { 0, 0, 87,0 }, //88
{ 0, 0, 88,0 }, { 0, 0, 89,0 }, { 0, 0, 90,0 }, { 0, 0, 91,0 }, //92
{ 0, 0, 92,0 }, { 0, 0, 93,0 }, { 0, 0, 94,0 }, { 0, 0, 95,0 }, //96
{ 0, 0, 96,0 }, { 0, 0, 97,0 }, { 0, 0, 98,0 }, { 0, 0, 99,0 }, //100
{ 0, 0,100,0 }, { 0, 0,101,0 }, { 0, 0,102,0 }, { 0, 0,103,0 }, //104
{ 0, 0,104,0 }, { 0, 0,105,0 }, { 0, 0,106,0 }, { 0, 0,107,0 }, //108
{ 0, 0,108,0 }, { 0, 0,109,0 }, { 0, 0,110,0 }, { 0, 0,111,0 }, //112
{ 0, 0,112,0 }, { 0, 0,113,0 }, { 0, 0,114,0 }, { 0, 0,115,0 }, //116
{ 0, 0,116,0 }, { 0, 0,117,0 }, { 0, 0,118,0 }, { 0, 0,119,0 }, //120
{ 0, 0,120,0 }, { 0, 0,121,0 }, { 0, 0,122,0 }, { 0, 0,123,0 }, //124
{ 0, 0,124,0 }, { 0, 0,125,0 }, { 0, 0,126,0 }, { 0, 0,127,0 }, //128
{ 0, 0,128,0 }, { 0, 0,129,0 }, { 0, 0,130,0 }, { 0, 0,131,0 }, //132
{ 0, 0,132,0 }, { 0, 0,133,0 }, { 0, 0,134,0 }, { 0, 0,135,0 }, //136
{ 0, 0,136,0 }, { 0, 0,137,0 }, { 0, 0,138,0 }, { 0, 0,139,0 }, //140
{ 0, 0,140,0 }, { 0, 0,141,0 }, { 0, 0,142,0 }, { 0, 0,143,0 }, //144
{ 0, 0,144,0 }, { 0, 0,145,0 }, { 0, 0,146,0 }, { 0, 0,147,0 }, //148
{ 0, 0,148,0 }, { 0, 0,149,0 }, { 0, 0,150,0 }, { 0, 0,151,0 }, //152
{ 0, 0,152,0 }, { 0, 0,153,0 }, { 0, 0,154,0 }, { 0, 0,155,0 }, //156
{ 0, 0,156,0 }, { 0, 0,157,0 }, { 0, 0,158,0 }, { 0, 0,159,0 }, //160
```

```
{   0,   0, 160, 0 }, {   0,   0, 161, 0 }, {   0,   0, 162, 0 }, {   0,   0, 163, 0 }, //164
{   0,   0, 164, 0 }, {   0,   0, 165, 0 }, {   0,   0, 166, 0 }, {   0,   0, 167, 0 }, //168
{   0,   0, 168, 0 }, {   0,   0, 169, 0 }, {   0,   0, 170, 0 }, {   0,   0, 171, 0 }, //172
{   0,   0, 172, 0 }, {   0,   0, 173, 0 }, {   0,   0, 174, 0 }, {   0,   0, 175, 0 }, //176
{   0,   0, 176, 0 }, {   0,   0, 177, 0 }, {   0,   0, 178, 0 }, {   0,   0, 179, 0 }, //180
{   0,   0, 180, 0 }, {   0,   0, 181, 0 }, {   0,   0, 182, 0 }, {   0,   0, 183, 0 }, //184
{   0,   0, 184, 0 }, {   0,   0, 185, 0 }, {   0,   0, 186, 0 }, {   0,   0, 187, 0 }, //188
{   0,   0, 188, 0 }, {   0,   0, 189, 0 }, {   0,   0, 190, 0 }, {   0,   0, 191, 0 }, //192
{   0,   0, 192, 0 }, {   0,   0, 193, 0 }, {   0,   0, 194, 0 }, {   0,   0, 195, 0 }, //196
{   0,   0, 196, 0 }, {   0,   0, 197, 0 }, {   0,   0, 198, 0 }, {   0,   0, 199, 0 }, //200
{   0,   0, 200, 0 }, {   0,   0, 201, 0 }, {   0,   0, 202, 0 }, {   0,   0, 203, 0 }, //204
{   0,   0, 204, 0 }, {   0,   0, 205, 0 }, {   0,   0, 206, 0 }, {   0,   0, 207, 0 }, //208
{   0,   0, 208, 0 }, {   0,   0, 209, 0 }, {   0,   0, 210, 0 }, {   0,   0, 211, 0 }, //212
{   0,   0, 212, 0 }, {   0,   0, 213, 0 }, {   0,   0, 214, 0 }, {   0,   0, 215, 0 }, //216
{   0,   0, 216, 0 }, {   0,   0, 217, 0 }, {   0,   0, 218, 0 }, {   0,   0, 219, 0 }, //220
{   0,   0, 220, 0 }, {   0,   0, 221, 0 }, {   0,   0, 222, 0 }, {   0,   0, 223, 0 }, //224
{   0,   0, 224, 0 }, {   0,   0, 225, 0 }, {   0,   0, 226, 0 }, {   0,   0, 227, 0 }, //228
{   0,   0, 228, 0 }, {   0,   0, 229, 0 }, {   0,   0, 230, 0 }, {   0,   0, 231, 0 }, //232
{   0,   0, 232, 0 }, {   0,   0, 233, 0 }, {   0,   0, 234, 0 }, {   0,   0, 235, 0 }, //236
{   0,   0, 236, 0 }, {   0,   0, 237, 0 }, {   0,   0, 238, 0 }, {   0,   0, 239, 0 }, //240
{   0,   0, 240, 0 }, {   0,   0, 241, 0 }, {   0,   0, 242, 0 }, {   0,   0, 243, 0 }, //244
{   0,   0, 244, 0 }, {   0,   0, 245, 0 }, {   0,   0, 246, 0 }, {   0,   0, 247, 0 }, //248
{   0,   0, 248, 0 }, {   0,   0, 249, 0 }, {   0,   0, 250, 0 }, {   0,   0, 251, 0 }, //252
{   0,   0, 252, 0 }, {   0,   0, 253, 0 }, {   0,   0, 254, 0 }, {   0,   0, 255, 0 }, //256
},
{  // 黄色饱和度编码
{   0,   0,   0, 0 }, {   1,   1,   0, 0 }, {   2,   2,   0, 0 }, {   3,   3,   0, 0 }, //4
{   4,   4,   0, 0 }, {   5,   5,   0, 0 }, {   6,   6,   0, 0 }, {   7,   7,   0, 0 }, //8
{   8,   8,   0, 0 }, {   9,   9,   0, 0 }, {  10,  10,   0, 0 }, {  11,  11,   0, 0 }, //12
{  12,  12,   0, 0 }, {  13,  13,   0, 0 }, {  14,  14,   0, 0 }, {  15,  15,   0, 0 }, //16
{  16,  16,   0, 0 }, {  17,  17,   0, 0 }, {  18,  18,   0, 0 }, {  19,  19,   0, 0 }, //20
{  20,  20,   0, 0 }, {  21,  21,   0, 0 }, {  22,  22,   0, 0 }, {  23,  23,   0, 0 }, //24
{  24,  24,   0, 0 }, {  25,  25,   0, 0 }, {  26,  26,   0, 0 }, {  27,  27,   0, 0 }, //28
{  28,  28,   0, 0 }, {  29,  29,   0, 0 }, {  30,  30,   0, 0 }, {  31,  31,   0, 0 }, //32
{  32,  32,   0, 0 }, {  33,  33,   0, 0 }, {  34,  34,   0, 0 }, {  35,  35,   0, 0 }, //36
{  36,  36,   0, 0 }, {  37,  37,   0, 0 }, {  38,  38,   0, 0 }, {  39,  39,   0, 0 }, //40
{  40,  40,   0, 0 }, {  41,  41,   0, 0 }, {  42,  42,   0, 0 }, {  43,  43,   0, 0 }, //44
{  44,  44,   0, 0 }, {  45,  45,   0, 0 }, {  46,  46,   0, 0 }, {  47,  47,   0, 0 }, //48
{  48,  48,   0, 0 }, {  49,  49,   0, 0 }, {  50,  50,   0, 0 }, {  51,  51,   0, 0 }, //52
{  52,  52,   0, 0 }, {  53,  53,   0, 0 }, {  54,  54,   0, 0 }, {  55,  55,   0, 0 }, //56
{  56,  56,   0, 0 }, {  57,  57,   0, 0 }, {  58,  58,   0, 0 }, {  59,  59,   0, 0 }, //60
{  60,  60,   0, 0 }, {  61,  61,   0, 0 }, {  62,  62,   0, 0 }, {  63,  63,   0, 0 }, //64
{  64,  64,   0, 0 }, {  65,  65,   0, 0 }, {  66,  66,   0, 0 }, {  67,  67,   0, 0 }, //68
{  68,  68,   0, 0 }, {  69,  69,   0, 0 }, {  70,  70,   0, 0 }, {  71,  71,   0, 0 }, //72
{  72,  72,   0, 0 }, {  73,  73,   0, 0 }, {  74,  74,   0, 0 }, {  75,  75,   0, 0 }, //76
{  76,  76,   0, 0 }, {  77,  77,   0, 0 }, {  78,  78,   0, 0 }, {  79,  79,   0, 0 }, //80
{  80,  80,   0, 0 }, {  81,  81,   0, 0 }, {  82,  82,   0, 0 }, {  83,  83,   0, 0 }, //84
{  84,  84,   0, 0 }, {  85,  85,   0, 0 }, {  86,  86,   0, 0 }, {  87,  87,   0, 0 }, //88
{  88,  88,   0, 0 }, {  89,  89,   0, 0 }, {  90,  90,   0, 0 }, {  91,  91,   0, 0 }, //92
{  92,  92,   0, 0 }, {  93,  93,   0, 0 }, {  94,  94,   0, 0 }, {  95,  95,   0, 0 }, //96
{  96,  96,   0, 0 }, {  97,  97,   0, 0 }, {  98,  98,   0, 0 }, {  99,  99,   0, 0 }, //100
```

```
{ 100,100,  0,0 }, { 101,101,  0,0 }, { 102,102,  0,0 }, { 103,103,  0,0 }, //104
{ 104,104,  0,0 }, { 105,105,  0,0 }, { 106,106,  0,0 }, { 107,107,  0,0 }, //108
{ 108,108,  0,0 }, { 109,109,  0,0 }, { 110,110,  0,0 }, { 111,111,  0,0 }, //112
{ 112,112,  0,0 }, { 113,113,  0,0 }, { 114,114,  0,0 }, { 115,115,  0,0 }, //116
{ 116,116,  0,0 }, { 117,117,  0,0 }, { 118,118,  0,0 }, { 119,119,  0,0 }, //120
{ 120,120,  0,0 }, { 121,121,  0,0 }, { 122,122,  0,0 }, { 123,123,  0,0 }, //124
{ 124,124,  0,0 }, { 125,125,  0,0 }, { 126,126,  0,0 }, { 127,127,  0,0 }, //128
{ 128,128,  0,0 }, { 129,129,  0,0 }, { 130,130,  0,0 }, { 131,131,  0,0 }, //132
{ 132,132,  0,0 }, { 133,133,  0,0 }, { 134,134,  0,0 }, { 135,135;  0,0 }, //136
{ 136,136,  0,0 }, { 137,137,  0,0 }, { 138,138,  0,0 }, { 139,139,  0,0 }, //140
{ 140,140,  0,0 }, { 141,141,  0,0 }, { 142,142,  0,0 }, { 143,143,  0,0 }, //144
{ 144,144,  0,0 }, { 145,145,  0,0 }, { 146,146,  0,0 }, { 147,147,  0,0 }, //148
{ 148,148,  0,0 }, { 149,149,  0,0 }, { 150,150,  0,0 }, { 151,151,  0,0 }, //152
{ 152,152,  0,0 }, { 153,153,  0,0 }, { 154,154,  0,0 }, { 155,155,  0,0 }, //156
{ 156,156,  0,0 }, { 157,157,  0,0 }, { 158,158,  0,0 }, { 159,159,  0,0 }, //160
{ 160,160,  0,0 }, { 161,161,  0,0 }, { 162,162,  0,0 }, { 163,163,  0,0 }, //164
{ 164,164,  0,0 }, { 165,165,  0,0 }, { 166,166,  0,0 }, { 167,167,  0,0 }, //168
{ 168,168,  0,0 }, { 169,169,  0,0 }, { 170,170,  0,0 }, { 171,171,  0,0 }, //172
{ 172,172,  0,0 }, { 173,173,  0,0 }, { 174,174,  0,0 }, { 175,175,  0,0 }, //176
{ 176,176,  0,0 }, { 177,177,  0,0 }, { 178,178,  0,0 }, { 179,179,  0,0 }, //180
{ 180,180,  0,0 }, { 181,181,  0,0 }, { 182,182,  0,0 }, { 183,183,  0,0 }, //184
{ 184,184,  0,0 }, { 185,185,  0,0 }, { 186,186,  0,0 }, { 187,187,  0,0 }, //188
{ 188,188,  0,0 }, { 189,189,  0,0 }, { 190,190,  0,0 }, { 191,191,  0,0 }, //192
{ 192,192,  0,0 }, { 193,193,  0,0 }, { 194,194,  0,0 }, { 195,195,  0,0 }, //196
{ 196,196,  0,0 }, { 197,197,  0,0 }, { 198,198,  0,0 }, { 199,199,  0,0 }, //200
{ 200,200,  0,0 }, { 201,201,  0,0 }, { 202,202,  0,0 }, { 203,203,  0,0 }, //204
{ 204,204,  0,0 }, { 205,205,  0,0 }, { 206,206,  0,0 }, { 207,207,  0,0 }, //208
{ 208,208,  0,0 }, { 209,209,  0,0 }, { 210,210,  0,0 }, { 211,211,  0,0 }, //212
{ 212,212,  0,0 }, { 213,213,  0,0 }, { 214,214,  0,0 }, { 215,215,  0,0 }, //216
{ 216,216,  0,0 }, { 217,217,  0,0 }, { 218,218,  0,0 }, { 219,219,  0,0 }, //220
{ 220,220,  0,0 }, { 221,221,  0,0 }, { 222,222,  0,0 }, { 223,223,  0,0 }, //224
{ 224,224,  0,0 }, { 225,225,  0,0 }, { 226,226,  0,0 }, { 227,227,  0,0 }, //228
{ 228,228,  0,0 }, { 229,229,  0,0 }, { 230,230,  0,0 }, { 231,231,  0,0 }, //232
{ 232,232,  0,0 }, { 233,233,  0,0 }, { 234,234,  0,0 }, { 235,235,  0,0 }, //236
{ 236,236,  0,0 }, { 237,237,  0,0 }, { 238,238,  0,0 }, { 239,239,  0,0 }, //240
{ 240,240,  0,0 }, { 241,241,  0,0 }, { 242,242,  0,0 }, { 243,243,  0,0 }, //244
{ 244,244,  0,0 }, { 245,245,  0,0 }, { 246,246,  0,0 }, { 247,247,  0,0 }, //248
{ 248,248,  0,0 }, { 249,249,  0,0 }, { 250,250,  0,0 }, { 251,251,  0,0 }, //252
{ 252,252,  0,0 }, { 253,253,  0,0 }, { 254,254,  0,0 }, { 255,255,  0,0 }, //256
},
{  // 青色饱和度编码
{  0,  0,  0,0 }, {  0,  1,  1,0 }, {  0,  2,  2,0 }, {  0,  3,  3,0 }, //4
{  0,  4,  4,0 }, {  0,  5,  5,0 }, {  0,  6,  6,0 }, {  0,  7,  7,0 }, //8
{  0,  8,  8,0 }, {  0,  9,  9,0 }, {  0, 10, 10,0 }, {  0, 11, 11,0 }, //12
{  0, 12, 12,0 }, {  0, 13, 13,0 }, {  0, 14, 14,0 }, {  0, 15, 15,0 }, //16
{  0, 16, 16,0 }, {  0, 17, 17,0 }, {  0, 18, 18,0 }, {  0, 19, 19,0 }, //20
{  0, 20, 20,0 }, {  0, 21, 21,0 }, {  0, 22, 22,0 }, {  0, 23, 23,0 }, //24
{  0, 24, 24,0 }, {  0, 25, 25,0 }, {  0, 26, 26,0 }, {  0, 27, 27,0 }, //28
{  0, 28, 28,0 }, {  0, 29, 29,0 }, {  0, 30, 30,0 }, {  0, 31, 31,0 }, //32
{  0, 32, 32,0 }, {  0, 33, 33,0 }, {  0, 34, 34,0 }, {  0, 35, 35,0 }, //36
{  0, 36, 36,0 }, {  0, 37, 37,0 }, {  0, 38, 38,0 }, {  0, 39, 39,0 }, //40
```

```
{ 0, 40, 40,0 }, { 0, 41, 41,0 }, { 0, 42, 42,0 }, { 0, 43, 43,0 }, //44
{ 0, 44, 44,0 }, { 0, 45, 45,0 }, { 0, 46, 46,0 }, { 0, 47, 47,0 }, //48
{ 0, 48, 48,0 }, { 0, 49, 49,0 }, { 0, 50, 50,0 }, { 0, 51, 51,0 }, //52
{ 0, 52, 52,0 }, { 0, 53, 53,0 }, { 0, 54, 54,0 }, { 0, 55, 55,0 }, //56
{ 0, 56, 56,0 }, { 0, 57, 57,0 }, { 0, 58, 58,0 }, { 0, 59, 59,0 }, //60
{ 0, 60, 60,0 }, { 0, 61, 61,0 }, { 0, 62, 62,0 }, { 0, 63, 63,0 }, //64
{ 0, 64, 64,0 }, { 0, 65, 65,0 }, { 0, 66, 66,0 }, { 0, 67, 67,0 }, //68
{ 0, 68, 68,0 }, { 0, 69, 69,0 }, { 0, 70, 70,0 }, { 0, 71, 71,0 }, //72
{ 0, 72, 72,0 }, { 0, 73, 73,0 }, { 0, 74, 74,0 }, { 0, 75, 75,0 }, //76
{ 0, 76, 76,0 }, { 0, 77, 77,0 }, { 0, 78, 78,0 }, { 0, 79, 79,0 }, //80
{ 0, 80, 80,0 }, { 0, 81, 81,0 }, { 0, 82, 82,0 }, { 0, 83, 83,0 }, //84
{ 0, 84, 84,0 }, { 0, 85, 85,0 }, { 0, 86, 86,0 }, { 0, 87, 87,0 }, //88
{ 0, 88, 88,0 }, { 0, 89, 89,0 }, { 0, 90, 90,0 }, { 0, 91, 91,0 }, //92
{ 0, 92, 92,0 }, { 0, 93, 93,0 }, { 0, 94, 94,0 }, { 0, 95, 95,0 }, //96
{ 0, 96, 96,0 }, { 0, 97, 97,0 }, { 0, 98, 98,0 }, { 0, 99, 99,0 }, //100
{ 0,100,100,0 }, { 0,101,101,0 }, { 0,102,102,0 }, { 0,103,103,0 }, //104
{ 0,104,104,0 }, { 0,105,105,0 }, { 0,106,106,0 }, { 0,107,107,0 }, //108
{ 0,108,108,0 }, { 0,109,109,0 }, { 0,110,110,0 }, { 0,111,111,0 }, //112
{ 0,112,112,0 }, { 0,113,113,0 }, { 0,114,114,0 }, { 0,115,115,0 }, //116
{ 0,116,116,0 }, { 0,117,117,0 }, { 0,118,118,0 }, { 0,119,119,0 }, //120
{ 0,120,120,0 }, { 0,121,121,0 }, { 0,122,122,0 }, { 0,123,123,0 }, //124
{ 0,124,124,0 }, { 0,125,125,0 }, { 0,126,126,0 }, { 0,127,127,0 }, //128
{ 0,128,128,0 }, { 0,129,129,0 }, { 0,130,130,0 }, { 0,131,131,0 }, //132
{ 0,132,132,0 }, { 0,133,133,0 }, { 0,134,134,0 }, { 0,135,135,0 }, //136
{ 0,136,136,0 }, { 0,137,137,0 }, { 0,138,138,0 }, { 0,139,139,0 }, //140
{ 0,140,140,0 }, { 0,141,141,0 }, { 0,142,142,0 }, { 0,143,143,0 }, //144
{ 0,144,144,0 }, { 0,145,145,0 }, { 0,146,146,0 }, { 0,147,147,0 }, //148
{ 0,148,148,0 }, { 0,149,149,0 }, { 0,150,150,0 }, { 0,151,151,0 }, //152
{ 0,152,152,0 }, { 0,153,153,0 }, { 0,154,154,0 }, { 0,155,155,0 }, //156
{ 0,156,156,0 }, { 0,157,157,0 }, { 0,158,158,0 }, { 0,159,159,0 }, //160
{ 0,160,160,0 }, { 0,161,161,0 }, { 0,162,162,0 }, { 0,163,163,0 }, //164
{ 0,164,164,0 }, { 0,165,165,0 }, { 0,166,166,0 }, { 0,167,167,0 }, //168
{ 0,168,168,0 }, { 0,169,169,0 }, { 0,170,170,0 }, { 0,171,171,0 }, //172
{ 0,172,172,0 }, { 0,173,173,0 }, { 0,174,174,0 }, { 0,175,175,0 }, //176
{ 0,176,176,0 }, { 0,177,177,0 }, { 0,178,178,0 }, { 0,179,179,0 }, //180
{ 0,180,180,0 }, { 0,181,181,0 }, { 0,182,182,0 }, { 0,183,183,0 }, //184
{ 0,184,184,0 }, { 0,185,185,0 }, { 0,186,186,0 }, { 0,187,187,0 }, //188
{ 0,188,188,0 }, { 0,189,189,0 }, { 0,190,190,0 }, { 0,191,191,0 }, //192
{ 0,192,192,0 }, { 0,193,193,0 }, { 0,194,194,0 }, { 0,195,195,0 }, //196
{ 0,196,196,0 }, { 0,197,197,0 }, { 0,198,198,0 }, { 0,199,199,0 }, //200
{ 0,200,200,0 }, { 0,201,201,0 }, { 0,202,202,0 }, { 0,203,203,0 }, //204
{ 0,204,204,0 }, { 0,205,205,0 }, { 0,206,206,0 }, { 0,207,207,0 }, //208
{ 0,208,208,0 }, { 0,209,209,0 }, { 0,210,210,0 }, { 0,211,211,0 }, //212
{ 0,212,212,0 }, { 0,213,213,0 }, { 0,214,214,0 }, { 0,215,215,0 }, //216
{ 0,216,216,0 }, { 0,217,217,0 }, { 0,218,218,0 }, { 0,219,219,0 }, //220
{ 0,220,220,0 }, { 0,221,221,0 }, { 0,222,222,0 }, { 0,223,223,0 }, //224
{ 0,224,224,0 }, { 0,225,225,0 }, { 0,226,226,0 }, { 0,227,227,0 }, //228
{ 0,228,228,0 }, { 0,229,229,0 }, { 0,230,230,0 }, { 0,231,231,0 }, //232
{ 0,232,232,0 }, { 0,233,233,0 }, { 0,234,234,0 }, { 0,235,235,0 }, //236
{ 0,236,236,0 }, { 0,237,237,0 }, { 0,238,238,0 }, { 0,239,239,0 }, //240
{ 0,240,240,0 }, { 0,241,241,0 }, { 0,242,242,0 }, { 0,243,243,0 }, //244
```

```
    {   0,244,244,0 },  {   0,245,245,0 },  {   0,246,246,0 },  {   0,247,247,0 },  //248
    {   0,248,248,0 },  {   0,249,249,0 },  {   0,250,250,0 },  {   0,251,251,0 },  //252
    {   0,252,252,0 },  {   0,253,253,0 },  {   0,254,254,0 },  {   0,255,255,0 },  //256
},
{   // 紫色饱和度编码
    {   0,  0,  0,0 },  {   1,  0,  1,0 },  {   2,  0,  2,0 },  {   3,  0,  3,0 },  //4
    {   4,  0,  4,0 },  {   5,  0,  5,0 },  {   6,  0,  6,0 },  {   7,  0,  7,0 },  //8
    {   8,  0,  8,0 },  {   9,  0,  9,0 },  {  10,  0, 10,0 },  {  11,  0, 11,0 },  //12
    {  12,  0, 12,0 },  {  13,  0, 13,0 },  {  14,  0, 14,0 },  {  15,  0, 15,0 },  //16
    {  16,  0, 16,0 },  {  17,  0, 17,0 },  {  18,  0, 18,0 },  {  19,  0, 19,0 },  //20
    {  20,  0, 20,0 },  {  21,  0, 21,0 },  {  22,  0, 22,0 },  {  23,  0, 23,0 },  //24
    {  24,  0, 24,0 },  {  25,  0, 25,0 },  {  26,  0, 26,0 },  {  27,  0, 27,0 },  //28
    {  28,  0, 28,0 },  {  29,  0, 29,0 },  {  30,  0, 30,0 },  {  31,  0, 31,0 },  //32
    {  32,  0, 32,0 },  {  33,  0, 33,0 },  {  34,  0, 34,0 },  {  35,  0, 35,0 },  //36
    {  36,  0, 36,0 },  {  37,  0, 37,0 },  {  38,  0, 38,0 },  {  39,  0, 39,0 },  //40
    {  40,  0, 40,0 },  {  41,  0, 41,0 },  {  42,  0, 42,0 },  {  43,  0, 43,0 },  //44
    {  44,  0, 44,0 },  {  45,  0, 45,0 },  {  46,  0, 46,0 },  {  47,  0, 47,0 },  //48
    {  48,  0, 48,0 },  {  49,  0, 49,0 },  {  50,  0, 50,0 },  {  51,  0, 51,0 },  //52
    {  52,  0, 52,0 },  {  53,  0, 53,0 },  {  54,  0, 54,0 },  {  55,  0, 55,0 },  //56
    {  56,  0, 56,0 },  {  57,  0, 57,0 },  {  58,  0, 58,0 },  {  59,  0, 59,0 },  //60
    {  60,  0, 60,0 },  {  61,  0, 61,0 },  {  62,  0, 62,0 },  {  63,  0, 63,0 },  //64
    {  64,  0, 64,0 },  {  65,  0, 65,0 },  {  66,  0, 66,0 },  {  67,  0, 67,0 },  //68
    {  68,  0, 68,0 },  {  69,  0, 69,0 },  {  70,  0, 70,0 },  {  71,  0, 71,0 },  //72
    {  72,  0, 72,0 },  {  73,  0, 73,0 },  {  74,  0, 74,0 },  {  75,  0, 75,0 },  //76
    {  76,  0, 76,0 },  {  77,  0, 77,0 },  {  78,  0, 78,0 },  {  79,  0, 79,0 },  //80
    {  80,  0, 80,0 },  {  81,  0, 81,0 },  {  82,  0, 82,0 },  {  83,  0, 83,0 },  //84
    {  84,  0, 84,0 },  {  85,  0, 85,0 },  {  86,  0, 86,0 },  {  87,  0, 87,0 },  //88
    {  88,  0, 88,0 },  {  89,  0, 89,0 },  {  90,  0, 90,0 },  {  91,  0, 91,0 },  //92
    {  92,  0, 92,0 },  {  93,  0, 93,0 },  {  94,  0, 94,0 },  {  95,  0, 95,0 },  //96
    {  96,  0, 96,0 },  {  97,  0, 97,0 },  {  98,  0, 98,0 },  {  99,  0, 99,0 },  //100
    { 100,  0,100,0 },  { 101,  0,101,0 },  { 102,  0,102,0 },  { 103,  0,103,0 },  //104
    { 104,  0,104,0 },  { 105,  0,105,0 },  { 106,  0,106,0 },  { 107,  0,107,0 },  //108
    { 108,  0,108,0 },  { 109,  0,109,0 },  { 110,  0,110,0 },  { 111,  0,111,0 },  //112
    { 112,  0,112,0 },  { 113,  0,113,0 },  { 114,  0,114,0 },  { 115,  0,115,0 },  //116
    { 116,  0,116,0 },  { 117,  0,117,0 },  { 118,  0,118,0 },  { 119,  0,119,0 },  //120
    { 120,  0,120,0 },  { 121,  0,121,0 },  { 122,  0,122,0 },  { 123,  0,123,0 },  //124
    { 124,  0,124,0 },  { 125,  0,125,0 },  { 126,  0,126,0 },  { 127,  0,127,0 },  //128
    { 128,  0,128,0 },  { 129,  0,129,0 },  { 130,  0,130,0 },  { 131,  0,131,0 },  //132
    { 132,  0,132,0 },  { 133,  0,133,0 },  { 134,  0,134,0 },  { 135,  0,135,0 },  //136
    { 136,  0,136,0 },  { 137,  0,137,0 },  { 138,  0,138,0 },  { 139,  0,139,0 },  //140
    { 140,  0,140,0 },  { 141,  0,141,0 },  { 142,  0,142,0 },  { 143,  0,143,0 },  //144
    { 144,  0,144,0 },  { 145,  0,145,0 },  { 146,  0,146,0 },  { 147,  0,147,0 },  //148
    { 148,  0,148,0 },  { 149,  0,149,0 },  { 150,  0,150,0 },  { 151,  0,151,0 },  //152
    { 152,  0,152,0 },  { 153,  0,153,0 },  { 154,  0,154,0 },  { 155,  0,155,0 },  //156
    { 156,  0,156,0 },  { 157,  0,157,0 },  { 158,  0,158,0 },  { 159,  0,159,0 },  //160
    { 160,  0,160,0 },  { 161,  0,161,0 },  { 162,  0,162,0 },  { 163,  0,163,0 },  //164
    { 164,  0,164,0 },  { 165,  0,165,0 },  { 166,  0,166,0 },  { 167,  0,167,0 },  //168
    { 168,  0,168,0 },  { 169,  0,169,0 },  { 170,  0,170,0 },  { 171,  0,171,0 },  //172
    { 172,  0,172,0 },  { 173,  0,173,0 },  { 174,  0,174,0 },  { 175,  0,175,0 },  //176
    { 176,  0,176,0 },  { 177,  0,177,0 },  { 178,  0,178,0 },  { 179,  0,179,0 },  //180
    { 180,  0,180,0 },  { 181,  0,181,0 },  { 182,  0,182,0 },  { 183,  0,183,0 },  //184
```

```
    { 184,    0, 184, 0 },  { 185,    0, 185, 0 },  { 186,    0, 186, 0 },  { 187,    0, 187, 0 },  //188
    { 188,    0, 188, 0 },  { 189,    0, 189, 0 },  { 190,    0, 190, 0 },  { 191,    0, 191, 0 },  //192
    { 192,    0, 192, 0 },  { 193,    0, 193, 0 },  { 194,    0, 194, 0 },  { 195,    0, 195, 0 },  //196
    { 196,    0, 196, 0 },  { 197,    0, 197, 0 },  { 198,    0, 198, 0 },  { 199,    0, 199, 0 },  //200
    { 200,    0, 200, 0 },  { 201,    0, 201, 0 },  { 202,    0, 202, 0 },  { 203,    0, 203, 0 },  //204
    { 204,    0, 204, 0 },  { 205,    0, 205, 0 },  { 206,    0, 206, 0 },  { 207,    0, 207, 0 },  //208
    { 208,    0, 208, 0 },  { 209,    0, 209, 0 },  { 210,    0, 210, 0 },  { 211,    0, 211, 0 },  //212
    { 212,    0, 212, 0 },  { 213,    0, 213, 0 },  { 214,    0, 214, 0 },  { 215,    0, 215, 0 },  //216
    { 216,    0, 216, 0 },  { 217,    0, 217, 0 },  { 218,    0, 218, 0 },  { 219,    0, 219, 0 },  //220
    { 220,    0, 220, 0 },  { 221,    0, 221, 0 },  { 222,    0, 222, 0 },  { 223,    0, 223, 0 },  //224
    { 224,    0, 224, 0 },  { 225,    0, 225, 0 },  { 226,    0, 226, 0 },  { 227,    0, 227, 0 },  //228
    { 228,    0, 228, 0 },  { 229,    0, 229, 0 },  { 230,    0, 230, 0 },  { 231,    0, 231, 0 },  //232
    { 232,    0, 232, 0 },  { 233,    0, 233, 0 },  { 234,    0, 234, 0 },  { 235,    0, 235, 0 },  //236
    { 236,    0, 236, 0 },  { 237,    0, 237, 0 },  { 238,    0, 238, 0 },  { 239,    0, 239, 0 },  //240
    { 240,    0, 240, 0 },  { 241,    0, 241, 0 },  { 242,    0, 242, 0 },  { 243,    0, 243, 0 },  //244
    { 244,    0, 244, 0 },  { 245,    0, 245, 0 },  { 246,    0, 246, 0 },  { 247,    0, 247, 0 },  //248
    { 248,    0, 248, 0 },  { 249,    0, 249, 0 },  { 250,    0, 250, 0 },  { 251,    0, 251, 0 },  //252
    { 252,    0, 252, 0 },  { 253,    0, 253, 0 },  { 254,    0, 254, 0 },  { 255,    0, 255, 0 },  //256
},
{   // 彩虹编码 1
    {   0,   0,   0, 0 },  {   0,   0,   7, 0 },  {   0,   0,  15, 0 },  {   0,   0,  23, 0 },  //4
    {   0,   0,  31, 0 },  {   0,   0,  39, 0 },  {   0,   0,  47, 0 },  {   0,   0,  55, 0 },  //8
    {   0,   0,  63, 0 },  {   0,   0,  71, 0 },  {   0,   0,  79, 0 },  {   0,   0,  87, 0 },  //12
    {   0,   0,  85, 0 },  {   0,   0, 103, 0 },  {   0,   0, 111, 0 },  {   0,   0, 119, 0 },  //16
    {   0,   0, 127, 0 },  {   0,   0, 135, 0 },  {   0,   0, 143, 0 },  {   0,   0, 151, 0 },  //20
    {   0,   0, 159, 0 },  {   0,   0, 167, 0 },  {   0,   0, 175, 0 },  {   0,   0, 183, 0 },  //24
    {   0,   0, 191, 0 },  {   0,   0, 199, 0 },  {   0,   0, 207, 0 },  {   0,   0, 215, 0 },  //28
    {   0,   0, 223, 0 },  {   0,   0, 231, 0 },  {   0,   0, 239, 0 },  {   0,   0, 247, 0 },  //32
    {   0,   0, 255, 0 },  {   0,   8, 255, 0 },  {   0,  16, 255, 0 },  {   0,  24, 255, 0 },  //36
    {   0,  32, 255, 0 },  {   0,  40, 255, 0 },  {   0,  48, 255, 0 },  {   0,  56, 255, 0 },  //40
    {   0,  64, 255, 0 },  {   0,  72, 255, 0 },  {   0,  80, 255, 0 },  {   0,  88, 255, 0 },  //44
    {   0,  96, 255, 0 },  {   0, 104, 255, 0 },  {   0, 112, 255, 0 },  {   0, 120, 255, 0 },  //48
    {   0, 128, 255, 0 },  {   0, 136, 255, 0 },  {   0, 144, 255, 0 },  {   0, 152, 255, 0 },  //52
    {   0, 160, 255, 0 },  {   0, 168, 255, 0 },  {   0, 176, 255, 0 },  {   0, 184, 255, 0 },  //56
    {   0, 192, 255, 0 },  {   0, 200, 255, 0 },  {   0, 208, 255, 0 },  {   0, 216, 255, 0 },  //60
    {   0, 224, 255, 0 },  {   6, 232, 255, 0 },  {   0, 240, 255, 0 },  {   0, 248, 255, 0 },  //64
    {   0, 255, 255, 0 },  {   0, 255, 247, 0 },  {   0, 255, 239, 0 },  {   0, 255, 231, 0 },  //68
    {   0, 255, 223, 0 },  {   0, 255, 215, 0 },  {   0, 255, 207, 0 },  {   0, 255, 199, 0 },  //72
    {   0, 255, 191, 0 },  {   0, 255, 183, 0 },  {   0, 255, 175, 0 },  {   0, 255, 167, 0 },  //76
    {   0, 255, 159, 0 },  {   0, 255, 151, 0 },  {   0, 255, 143, 0 },  {   0, 255, 135, 0 },  //80
    {   0, 255, 127, 0 },  {   0, 255, 119, 0 },  {   0, 255, 111, 0 },  {   0, 255, 103, 0 },  //84
    {   0, 255,  95, 0 },  {   0, 255,  87, 0 },  {   0, 255,  79, 0 },  {   0, 255,  71, 0 },  //88
    {   0, 255,  63, 0 },  {   0, 255,  55, 0 },  {   0, 255,  47, 0 },  {   0, 255,  39, 0 },  //92
    {   0, 255,  31, 0 },  {   0, 255,  23, 0 },  {   0, 255,  15, 0 },  {   0, 255,   7, 0 },  //96
    {   0, 255,   0, 0 },  {   8, 255,   0, 0 },  {  16, 255,   0, 0 },  {  24, 255,   0, 0 },  //100
    {  32, 255,   0, 0 },  {  40, 255,   0, 0 },  {  48, 255,   0, 0 },  {  56, 255,   0, 0 },  //104
    {  64, 255,   0, 0 },  {  72, 255,   0, 0 },  {  80, 255,   0, 0 },  {  88, 255,   0, 0 },  //108
    {  96, 255,   0, 0 },  { 104, 255,   0, 0 },  { 112, 255,   0, 0 },  { 120, 255,   0, 0 },  //112
    { 128, 255,   0, 0 },  { 136, 255,   0, 0 },  { 144, 255,   0, 0 },  { 152, 255,   0, 0 },  //116
    { 160, 255,   0, 0 },  { 168, 255,   0, 0 },  { 176, 255,   0, 0 },  { 184, 255,   0, 0 },  //120
    { 192, 255,   0, 0 },  { 200, 255,   0, 0 },  { 208, 255,   0, 0 },  { 216, 255,   0, 0 },  //124
```

```
{ 224, 255,   0, 0 }, { 232, 255,   0, 0 }, { 240, 255,   0, 0 }, { 248, 255,   0, 0 }, //128
{ 255, 255,   0, 0 }, { 255, 251,   0, 0 }, { 255, 247,   0, 0 }, { 255, 243,   0, 0 }, //132
{ 255, 239,   0, 0 }, { 255, 235,   0, 0 }, { 255, 231,   0, 0 }, { 255, 227,   0, 0 }, //136
{ 255, 223,   0, 0 }, { 255, 219,   0, 0 }, { 255, 215,   0, 0 }, { 255, 211,   0, 0 }, //140
{ 255, 207,   0, 0 }, { 255, 203,   0, 0 }, { 255, 199,   0, 0 }, { 255, 195,   0, 0 }, //144
{ 255, 191,   0, 0 }, { 255, 187,   0, 0 }, { 255, 183,   0, 0 }, { 255, 179,   0, 0 }, //148
{ 255, 175,   0, 0 }, { 255, 171,   0, 0 }, { 255, 167,   0, 0 }, { 255, 163,   0, 0 }, //152
{ 255, 159,   0, 0 }, { 255, 155,   0, 0 }, { 255, 151,   0, 0 }, { 255, 147,   0, 0 }, //156
{ 255, 143,   0, 0 }, { 255, 139,   0, 0 }, { 255, 135,   0, 0 }, { 255, 131,   0, 0 }, //160
{ 255, 127,   0, 0 }, { 255, 123,   0, 0 }, { 255, 119,   0, 0 }, { 255, 115,   0, 0 }, //164
{ 255, 111,   0, 0 }, { 255, 107,   0, 0 }, { 255, 103,   0, 0 }, { 255,  99,   0, 0 }, //168
{ 255,  95,   0, 0 }, { 255,  91,   0, 0 }, { 255,  87,   0, 0 }, { 255,  83,   0, 0 }, //172
{ 255,  79,   0, 0 }, { 255,  75,   0, 0 }, { 255,  71,   0, 0 }, { 255,  67,   0, 0 }, //176
{ 255,  63,   0, 0 }, { 255,  59,   0, 0 }, { 255,  55,   0, 0 }, { 255,  51,   0, 0 }, //180
{ 255,  47,   0, 0 }, { 255,  43,   0, 0 }, { 255,  39,   0, 0 }, { 255,  35,   0, 0 }, //184
{ 255,  31,   0, 0 }, { 255,  27,   0, 0 }, { 255,  23,   0, 0 }, { 255,  19,   0, 0 }, //188
{ 255,  15,   0, 0 }, { 255,  11,   0, 0 }, { 255,   7,   0, 0 }, { 255,   3,   0, 0 }, //192
{ 255,   0,   0, 0 }, { 255,   4,   4, 0 }, { 255,   8,   8, 0 }, { 255,  12,  12, 0 }, //196
{ 255,  16,  16, 0 }, { 255,  20,  20, 0 }, { 255,  24,  24, 0 }, { 255,  28,  28, 0 }, //200
{ 255,  32,  32, 0 }, { 255,  36,  36, 0 }, { 255,  40,  40, 0 }, { 255,  44,  44, 0 }, //204
{ 255,  48,  48, 0 }, { 255,  52,  52, 0 }, { 255,  56,  56, 0 }, { 255,  60,  60, 0 }, //208
{ 255,  64,  64, 0 }, { 255,  68,  68, 0 }, { 255,  72,  72, 0 }, { 255,  76,  76, 0 }, //212
{ 255,  80,  80, 0 }, { 255,  84,  84, 0 }, { 255,  88,  88, 0 }, { 255,  92,  92, 0 }, //216
{ 255,  96,  96, 0 }, { 255, 100, 100, 0 }, { 255, 104, 104, 0 }, { 255, 108, 108, 0 }, //220
{ 255, 112, 112, 0 }, { 255, 116, 116, 0 }, { 255, 120, 120, 0 }, { 255, 124, 124, 0 }, //224
{ 255, 128, 128, 0 }, { 255, 132, 132, 0 }, { 255, 136, 136, 0 }, { 255, 140, 140, 0 }, //228
{ 255, 144, 144, 0 }, { 255, 148, 148, 0 }, { 255, 152, 152, 0 }, { 255, 156, 156, 0 }, //232
{ 255, 160, 160, 0 }, { 255, 164, 164, 0 }, { 255, 168, 168, 0 }, { 255, 172, 172, 0 }, //236
{ 255, 176, 176, 0 }, { 255, 180, 180, 0 }, { 255, 184, 184, 0 }, { 255, 188, 188, 0 }, //240
{ 255, 192, 192, 0 }, { 255, 196, 196, 0 }, { 255, 200, 200, 0 }, { 255, 204, 204, 0 }, //244
{ 255, 208, 208, 0 }, { 255, 212, 212, 0 }, { 255, 216, 216, 0 }, { 255, 220, 220, 0 }, //248
{ 255, 224, 224, 0 }, { 255, 228, 228, 0 }, { 255, 232, 232, 0 }, { 255, 236, 236, 0 }, //252
{ 255, 240, 240, 0 }, { 255, 244, 244, 0 }, { 255, 248, 248, 0 }, { 255, 252, 252, 0 }, //256
},
{   // 彩虹编码 2
{   0,   0, 255, 0 }, {   0,   3, 255, 0 }, {   0,   7, 255, 0 }, {   0,  11, 255, 0 }, //4
{   0,  15, 255, 0 }, {   0,  19, 255, 0 }, {   0,  23, 255, 0 }, {   0,  27, 255, 0 }, //8
{   0,  31, 255, 0 }, {   0,  35, 255, 0 }, {   0,  39, 255, 0 }, {   0,  43, 255, 0 }, //12
{   0,  47, 255, 0 }, {   0,  51, 255, 0 }, {   0,  55, 255, 0 }, {   0,  59, 255, 0 }, //16
{   0,  63, 255, 0 }, {   0,  67, 255, 0 }, {   0,  71, 255, 0 }, {   0,  75, 255, 0 }, //20
{   0,  79, 255, 0 }, {   0,  83, 255, 0 }, {   0,  87, 255, 0 }, {   0,  91, 255, 0 }, //24
{   0,  95, 255, 0 }, {   0,  99, 255, 0 }, {   0, 103, 255, 0 }, {   0, 107, 255, 0 }, //28
{   0, 111, 255, 0 }, {   0, 115, 255, 0 }, {   0, 119, 255, 0 }, {   0, 123, 255, 0 }, //32
{   0, 127, 255, 0 }, {   0, 131, 255, 0 }, {   0, 135, 255, 0 }, {   0, 139, 255, 0 }, //36
{   0, 143, 255, 0 }, {   0, 147, 255, 0 }, {   0, 151, 255, 0 }, {   0, 155, 255, 0 }, //40
{   0, 159, 255, 0 }, {   0, 163, 255, 0 }, {   0, 167, 255, 0 }, {   0, 171, 255, 0 }, //44
{   0, 175, 255, 0 }, {   0, 179, 255, 0 }, {   0, 183, 255, 0 }, {   0, 187, 255, 0 }, //48
{   0, 191, 255, 0 }, {   0, 195, 255, 0 }, {   0, 199, 255, 0 }, {   0, 203, 255, 0 }, //52
{   0, 207, 255, 0 }, {   0, 211, 255, 0 }, {   0, 215, 255, 0 }, {   0, 219, 255, 0 }, //56
{   0, 223, 255, 0 }, {   0, 227, 255, 0 }, {   0, 231, 255, 0 }, {   0, 235, 255, 0 }, //60
{   0, 239, 255, 0 }, {   6, 243, 255, 0 }, {   0, 247, 255, 0 }, {   0, 251, 255, 0 }, //64
```

```
{   0,255,255,0 }, {   0,255,247,0 }, {   0,255,239,0 }, {   0,255,231,0 }, //68
{   0,255,223,0 }, {   0,255,215,0 }, {   0,255,207,0 }, {   0,255,199,0 }, //72
{   0,255,191,0 }, {   0,255,183,0 }, {   0,255,175,0 }, {   0,255,167,0 }, //76
{   0,255,159,0 }, {   0,255,151,0 }, {   0,255,143,0 }, {   0,255,135,0 }, //80
{   0,255,127,0 }, {   0,255,119,0 }, {   0,255,111,0 }, {   0,255,103,0 }, //84
{   0,255, 95,0 }, {   0,255, 87,0 }, {   0,255, 79,0 }, {   0,255, 71,0 }, //88
{   0,255, 63,0 }, {   0,255, 55,0 }, {   0,255, 47,0 }, {   0,255, 39,0 }, //92
{   0,255, 31,0 }, {   0,255, 23,0 }, {   0,255, 15,0 }, {   0,255,  7,0 }, //96
{   0,255,  0,0 }, {   8,255,  0,0 }, {  16,255,  0,0 }, {  24,255,  0,0 }, //100
{  32,255,  0,0 }, {  40,255,  0,0 }, {  48,255,  0,0 }, {  56,255,  0,0 }, //104
{  64,255,  0,0 }, {  72,255,  0,0 }, {  80,255,  0,0 }, {  88,255,  0,0 }, //108
{  96,255,  0,0 }, { 104,255,  0,0 }, { 112,255,  0,0 }, { 120,255,  0,0 }, //112
{ 128,255,  0,0 }, { 136,255,  0,0 }, { 144,255,  0,0 }, { 152,255,  0,0 }, //116
{ 160,255,  0,0 }, { 168,255,  0,0 }, { 176,255,  0,0 }, { 184,255,  0,0 }, //120
{ 192,255,  0,0 }, { 200,255,  0,0 }, { 208,255,  0,0 }, { 216,255,  0,0 }, //124
{ 224,255,  0,0 }, { 232,255,  0,0 }, { 240,255,  0,0 }, { 248,255,  0,0 }, //128
{ 255,255,  0,0 }, { 255,251,  0,0 }, { 255,247,  0,0 }, { 255,243,  0,0 }, //132
{ 255,239,  0,0 }, { 255,235,  0,0 }, { 255,231,  0,0 }, { 255,227,  0,0 }, //136
{ 255,223,  0,0 }, { 255,219,  0,0 }, { 255,215,  0,0 }, { 255,211,  0,0 }, //140
{ 255,207,  0,0 }, { 255,203,  0,0 }, { 255,199,  0,0 }, { 255,195,  0,0 }, //144
{ 255,191,  0,0 }, { 255,187,  0,0 }, { 255,183,  0,0 }, { 255,179,  0,0 }, //148
{ 255,175,  0,0 }, { 255,171,  0,0 }, { 255,167,  0,0 }, { 255,163,  0,0 }, //152
{ 255,159,  0,0 }, { 255,155,  0,0 }, { 255,151,  0,0 }, { 255,147,  0,0 }, //156
{ 255,143,  0,0 }, { 255,139,  0,0 }, { 255,135,  0,0 }, { 255,131,  0,0 }, //160
{ 255,127,  0,0 }, { 255,123,  0,0 }, { 255,119,  0,0 }, { 255,115,  0,0 }, //164
{ 255,111,  0,0 }, { 255,107,  0,0 }, { 255,103,  0,0 }, { 255, 99,  0,0 }, //168
{ 255, 95,  0,0 }, { 255, 91,  0,0 }, { 255, 87,  0,0 }, { 255, 83,  0,0 }, //172
{ 255, 79,  0,0 }, { 255, 75,  0,0 }, { 255, 71,  0,0 }, { 255, 67,  0,0 }, //176
{ 255, 63,  0,0 }, { 255, 59,  0,0 }, { 255, 55,  0,0 }, { 255, 51,  0,0 }, //180
{ 255, 47,  0,0 }, { 255, 43,  0,0 }, { 255, 39,  0,0 }, { 255, 35,  0,0 }, //184
{ 255, 31,  0,0 }, { 255, 27,  0,0 }, { 255, 23,  0,0 }, { 255, 19,  0,0 }, //188
{ 255, 15,  0,0 }, { 255, 11,  0,0 }, { 255,  7,  0,0 }, { 255,  3,  0,0 }, //192
{ 255,  0,  0,0 }, { 255,  4,  4,0 }, { 255,  8,  8,0 }, { 255, 12, 12,0 }, //196
{ 255, 16, 16,0 }, { 255, 20, 20,0 }, { 255, 24, 24,0 }, { 255, 28, 28,0 }, //200
{ 255, 32, 32,0 }, { 255, 36, 36,0 }, { 255, 40, 40,0 }, { 255, 44, 44,0 }, //204
{ 255, 48, 48,0 }, { 255, 52, 52,0 }, { 255, 56, 56,0 }, { 255, 60, 60,0 }, //208
{ 255, 64, 64,0 }, { 255, 68, 68,0 }, { 255, 72, 72,0 }, { 255, 76, 76,0 }, //212
{ 255, 80, 80,0 }, { 255, 84, 84,0 }, { 255, 88, 88,0 }, { 255, 92, 92,0 }, //216
{ 255, 96, 96,0 }, { 255,100,100,0 }, { 255,104,104,0 }, { 255,108,108,0 }, //220
{ 255,112,112,0 }, { 255,116,116,0 }, { 255,120,120,0 }, { 255,124,124,0 }, //224
{ 255,128,128,0 }, { 255,132,132,0 }, { 255,136,136,0 }, { 255,140,140,0 }, //228
{ 255,144,144,0 }, { 255,148,148,0 }, { 255,152,152,0 }, { 255,156,156,0 }, //232
{ 255,160,160,0 }, { 255,164,164,0 }, { 255,168,168,0 }, { 255,172,172,0 }, //236
{ 255,176,176,0 }, { 255,180,180,0 }, { 255,184,184,0 }, { 255,188,188,0 }, //240
{ 255,192,192,0 }, { 255,196,196,0 }, { 255,200,200,0 }, { 255,204,204,0 }, //244
{ 255,208,208,0 }, { 255,212,212,0 }, { 255,216,216,0 }, { 255,220,220,0 }, //248
{ 255,224,224,0 }, { 255,228,228,0 }, { 255,232,232,0 }, { 255,236,236,0 }, //252
{ 255,240,240,0 }, { 255,244,244,0 }, { 255,248,248,0 }, { 255,252,252,0 }, //256
},
{   // 热金属编码 1
{   0,  0,  0,0 }, {   0,  0,  4,0 }, {   0,  0,  8,0 }, {   0,  0, 12,0 }, //4
```

```
{   0,   0,  16,0 }, {   0,   0,  20,0 }, {   0,   0,  24,0 }, {   0,   0,  28,0 }, //8
{   0,   0,  32,0 }, {   0,   0,  36,0 }, {   0,   0,  40,0 }, {   0,   0,  44,0 }, //12
{   0,   0,  48,0 }, {   0,   0,  52,0 }, {   0,   0,  56,0 }, {   0,   0,  60,0 }, //16
{   0,   0,  64,0 }, {   0,   0,  68,0 }, {   0,   0,  72,0 }, {   0,   0,  76,0 }, //20
{   0,   0,  80,0 }, {   0,   0,  84,0 }, {   0,   0,  88,0 }, {   0,   0,  92,0 }, //24
{   0,   0,  96,0 }, {   0,   0, 100,0 }, {   0,   0, 104,0 }, {   0,   0, 108,0 }, //28
{   0,   0, 112,0 }, {   0,   0, 116,0 }, {   0,   0, 120,0 }, {   0,   0, 124,0 }, //32
{   0,   0, 128,0 }, {   0,   0, 132,0 }, {   0,   0, 136,0 }, {   0,   0, 140,0 }, //36
{   0,   0, 144,0 }, {   0,   0, 148,0 }, {   0,   0, 152,0 }, {   0,   0, 156,0 }, //40
{   0,   0, 160,0 }, {   0,   0, 164,0 }, {   0,   0, 168,0 }, {   0,   0, 172,0 }, //44
{   0,   0, 176,0 }, {   0,   0, 180,0 }, {   0,   0, 184,0 }, {   0,   0, 188,0 }, //48
{   0,   0, 192,0 }, {   0,   0, 196,0 }, {   0,   0, 200,0 }, {   0,   0, 204,0 }, //52
{   0,   0, 208,0 }, {   0,   0, 212,0 }, {   0,   0, 216,0 }, {   0,   0, 220,0 }, //56
{   0,   0, 224,0 }, {   0,   0, 228,0 }, {   0,   0, 232,0 }, {   0,   0, 236,0 }, //60
{   0,   0, 240,0 }, {   0,   0, 244,0 }, {   0,   0, 248,0 }, {   0,   0, 252,0 }, //64
{   0,   0, 255,0 }, {   4,   0, 255,0 }, {   8,   0, 255,0 }, {  12,   0, 255,0 }, //68
{  16,   0, 255,0 }, {  20,   0, 255,0 }, {  24,   0, 255,0 }, {  28,   0, 255,0 }, //72
{  32,   0, 255,0 }, {  36,   0, 255,0 }, {  40,   0, 255,0 }, {  44,   0, 255,0 }, //76
{  48,   0, 255,0 }, {  52,   0, 255,0 }, {  56,   0, 255,0 }, {  60,   0, 255,0 }, //80
{  64,   0, 255,0 }, {  68,   0, 255,0 }, {  72,   0, 255,0 }, {  76,   0, 255,0 }, //84
{  80,   0, 255,0 }, {  84,   0, 255,0 }, {  88,   0, 255,0 }, {  92,   0, 255,0 }, //88
{  96,   0, 255,0 }, { 100,   0, 255,0 }, { 104,   0, 255,0 }, { 108,   0, 255,0 }, //92
{ 112,   0, 255,0 }, { 116,   0, 255,0 }, { 120,   0, 255,0 }, { 124,   0, 255,0 }, //96
{ 128,   0, 247,0 }, { 132,   0, 239,0 }, { 136,   0, 231,0 }, { 140,   0, 223,0 }, //100
{ 144,   0, 215,0 }, { 148,   0, 207,0 }, { 152,   0, 199,0 }, { 156,   0, 191,0 }, //104
{ 160,   0, 183,0 }, { 164,   0, 175,0 }, { 168,   0, 167,0 }, { 172,   0, 159,0 }, //108
{ 176,   0, 151,0 }, { 180,   0, 143,0 }, { 184,   0, 135,0 }, { 188,   0, 127,0 }, //112
{ 192,   0, 119,0 }, { 196,   0, 111,0 }, { 200,   0, 103,0 }, { 204,   0,  95,0 }, //116
{ 208,   0,  87,0 }, { 212,   0,  79,0 }, { 216,   0,  71,0 }, { 220,   0,  63,0 }, //120
{ 224,   0,  55,0 }, { 228,   0,  47,0 }, { 232,   0,  39,0 }, { 236,   0,  31,0 }, //124
{ 240,   0,  23,0 }, { 244,   0,  15,0 }, { 248,   0,   7,0 }, { 252,   0,   0,0 }, //128
{ 255,   0,   0,0 }, { 255,   4,   0,0 }, { 255,   8,   0,0 }, { 255,  12,   0,0 }, //132
{ 255,  16,   0,0 }, { 255,  20,   0,0 }, { 255,  24,   0,0 }, { 255,  28,   0,0 }, //136
{ 255,  32,   0,0 }, { 255,  36,   0,0 }, { 255,  40,   0,0 }, { 255,  44,   0,0 }, //140
{ 255,  48,   0,0 }, { 255,  52,   0,0 }, { 255,  56,   0,0 }, { 255,  60,   0,0 }, //144
{ 255,  64,   0,0 }, { 255,  68,   0,0 }, { 255,  72,   0,0 }, { 255,  76,   0,0 }, //148
{ 255,  80,   0,0 }, { 255,  84,   0,0 }, { 255,  88,   0,0 }, { 255,  92,   0,0 }, //152
{ 255,  96,   0,0 }, { 255, 100,   0,0 }, { 255, 104,   0,0 }, { 255, 108,   0,0 }, //156
{ 255, 112,   0,0 }, { 255, 116,   0,0 }, { 255, 120,   0,0 }, { 255, 124,   0,0 }, //160
{ 255, 128,   0,0 }, { 255, 132,   0,0 }, { 255, 136,   0,0 }, { 255, 140,   0,0 }, //164
{ 255, 144,   0,0 }, { 255, 148,   0,0 }, { 255, 152,   0,0 }, { 255, 156,   0,0 }, //168
{ 255, 160,   0,0 }, { 255, 164,   0,0 }, { 255, 168,   0,0 }, { 255, 172,   0,0 }, //172
{ 255, 176,   0,0 }, { 255, 180,   0,0 }, { 255, 184,   0,0 }, { 255, 188,   0,0 }, //176
{ 255, 192,   0,0 }, { 255, 196,   0,0 }, { 255, 200,   0,0 }, { 255, 204,   0,0 }, //180
{ 255, 208,   0,0 }, { 255, 212,   0,0 }, { 255, 216,   0,0 }, { 255, 220,   0,0 }, //184
{ 255, 224,   0,0 }, { 255, 228,   0,0 }, { 255, 232,   0,0 }, { 255, 236,   0,0 }, //188
{ 255, 240,   0,0 }, { 255, 244,   0,0 }, { 255, 248,   0,0 }, { 255, 252,   0,0 }, //192
{ 255, 255,   0,0 }, { 255, 255,   4,0 }, { 255, 255,   8,0 }, { 255, 255,  12,0 }, //196
{ 255, 255,  16,0 }, { 255, 255,  20,0 }, { 255, 255,  24,0 }, { 255, 255,  28,0 }, //200
{ 255, 255,  32,0 }, { 255, 255,  36,0 }, { 255, 255,  40,0 }, { 255, 255,  44,0 }, //204
{ 255, 255,  48,0 }, { 255, 255,  52,0 }, { 255, 255,  56,0 }, { 255, 255,  60,0 }, //208
```

```
{ 255,255, 64,0 }, { 255,255, 68,0 }, { 255,255, 72,0 }, { 255,255, 76,0 }, //212
{ 255,255, 80,0 }, { 255,255, 84,0 }, { 255,255, 88,0 }, { 255,255, 92,0 }, //216
{ 255,255, 96,0 }, { 255,255,100,0 }, { 255,255,104,0 }, { 255,255,108,0 }, //220
{ 255,255,112,0 }, { 255,255,116,0 }, { 255,255,120,0 }, { 255,255,124,0 }, //224
{ 255,255,128,0 }, { 255,255,132,0 }, { 255,255,136,0 }, { 255,255,140,0 }, //228
{ 255,255,144,0 }, { 255,255,148,0 }, { 255,255,152,0 }, { 255,255,156,0 }, //232
{ 255,255,160,0 }, { 255,255,164,0 }, { 255,255,168,0 }, { 255,255,172,0 }, //236
{ 255,255,176,0 }, { 255,255,180,0 }, { 255,255,184,0 }, { 255,255,188,0 }, //240
{ 255,255,192,0 }, { 255,255,196,0 }, { 255,255,200,0 }, { 255,255,204,0 }, //244
{ 255,255,208,0 }, { 255,255,212,0 }, { 255,255,216,0 }, { 255,255,220,0 }, //248
{ 255,255,224,0 }, { 255,255,228,0 }, { 255,255,232,0 }, { 255,255,236,0 }, //252
{ 255,255,240,0 }, { 255,255,244,0 }, { 255,255,248,0 }, { 255,255,252,0 }, //256
},
{  // 热金属编码 2
{   0,  0,  0,0 }, {   0,  0, 30,0 }, {   0,  0, 34,0 }, {   0,  0, 38,0 }, //4
{   0,  0, 43,0 }, {   0,  0, 47,0 }, {   0,  0, 51,0 }, {   0,  0, 55,0 }, //8
{   0,  0, 60,0 }, {   0,  0, 64,0 }, {   0,  0, 68,0 }, {   0,  0, 72,0 }, //12
{   0,  0, 77,0 }, {   0,  0, 81,0 }, {   0,  0, 85,0 }, {   0,  0, 89,0 }, //16
{   0,  0, 94,0 }, {   3,  0, 98,0 }, {   6,  0,102,0 }, {   9,  0,106,0 }, //20
{  12,  0,111,0 }, {  15,  0,115,0 }, {  18,  0,119,0 }, {  21,  0,123,0 }, //24
{  24,  0,128,0 }, {  27,  0,132,0 }, {  30,  0,136,0 }, {  33,  0,140,0 }, //28
{  36,  0,145,0 }, {  39,  0,149,0 }, {  42,  0,153,0 }, {  45,  0,157,0 }, //32
{  48,  0,162,0 }, {  51,  0,166,0 }, {  54,  0,169,0 }, {  57,  0,173,0 }, //36
{  60,  0,177,0 }, {  63,  0,180,0 }, {  66,  0,184,0 }, {  69,  0,187,0 }, //40
{  72,  0,190,0 }, {  75,  0,194,0 }, {  78,  0,197,0 }, {  81,  0,200,0 }, //44
{  84,  0,204,0 }, {  87,  0,207,0 }, {  90,  0,210,0 }, {  93,  0,214,0 }, //48
{  96,  0,217,0 }, {  99,  0,220,0 }, { 102,  0,223,0 }, { 105,  0,227,0 }, //52
{ 108,  0,230,0 }, { 111,  0,229,0 }, { 114,  0,228,0 }, { 117,  0,228,0 }, //56
{ 120,  0,227,0 }, { 123,  0,226,0 }, { 126,  0,226,0 }, { 129,  0,225,0 }, //60
{ 132,  0,224,0 }, { 135,  0,223,0 }, { 138,  0,219,0 }, { 141,  0,216,0 }, //64
{ 144,  0,212,0 }, { 147,  0,208,0 }, { 150,  0,205,0 }, { 153,  0,201,0 }, //68
{ 156,  0,197,0 }, { 157,  0,194,0 }, { 159,  0,190,0 }, { 160,  0,186,0 }, //72
{ 162,  0,183,0 }, { 163,  0,179,0 }, { 165,  0,175,0 }, { 166,  0,172,0 }, //76
{ 167,  0,168,0 }, { 168,  0,164,0 }, { 170,  0,161,0 }, { 171,  0,157,0 }, //80
{ 173,  0,153,0 }, { 174,  0,150,0 }, { 176,  0,146,0 }, { 177,  0,142,0 }, //84
{ 178,  0,139,0 }, { 179,  0,135,0 }, { 181,  0,131,0 }, { 182,  0,127,0 }, //88
{ 184,  0,124,0 }, { 185,  0,120,0 }, { 187,  0,116,0 }, { 188,  0,113,0 }, //92
{ 189,  0,109,0 }, { 190,  0,105,0 }, { 192,  0,102,0 }, { 193,  0, 98,0 }, //96
{ 195,  0, 94,0 }, { 196,  0, 91,0 }, { 198,  0, 87,0 }, { 199,  0, 83,0 }, //100
{ 200,  0, 80,0 }, { 201,  0, 77,0 }, { 203,  3, 73,0 }, { 204,  6, 69,0 }, //104
{ 206,  8, 66,0 }, { 207, 11, 62,0 }, { 209, 14, 58,0 }, { 210, 17, 55,0 }, //108
{ 211, 19, 51,0 }, { 212, 22, 47,0 }, { 214, 25, 44,0 }, { 215, 28, 40,0 }, //112
{ 217, 30, 37,0 }, { 218, 33, 34,0 }, { 220, 36, 30,0 }, { 221, 39, 26,0 }, //116
{ 222, 41, 22,0 }, { 223, 44, 18,0 }, { 225, 47, 15,0 }, { 226, 50, 11,0 }, //120
{ 228, 52,  7,0 }, { 229, 55,  4,0 }, { 231, 58,  0,0 }, { 232, 61,  0,0 }, //124
{ 233, 63,  0,0 }, { 234, 66,  0,0 }, { 236, 69,  0,0 }, { 237, 72,  0,0 }, //128
{ 239, 74,  0,0 }, { 240, 77,  0,0 }, { 242, 80,  0,0 }, { 243, 83,  0,0 }, //132
{ 244, 85,  0,0 }, { 245, 88,  0,0 }, { 247, 91,  0,0 }, { 248, 94,  0,0 }, //136
{ 250, 96,  0,0 }, { 252, 99,  0,0 }, { 253,102,  0,0 }, { 254,105,  0,0 }, //140
{ 255,107,  0,0 }, { 255,109,  0,0 }, { 255,111,  0,0 }, { 255,112,  0,0 }, //144
{ 255,114,  0,0 }, { 255,116,  0,0 }, { 255,118,  0,0 }, { 255,119,  0,0 }, //148
```

```
{ 255, 121,   0, 0 },  { 255, 123,   0, 0 },  { 255, 125,   0, 0 },  { 255, 126,   0, 0 },  //152
{ 255, 128,   0, 0 },  { 255, 130,   0, 0 },  { 255, 132,   0, 0 },  { 255, 133,   0, 0 },  //156
{ 255, 135,   0, 0 },  { 255, 137,   0, 0 },  { 255, 139,   0, 0 },  { 255, 140,   0, 0 },  //160
{ 255, 142,   0, 0 },  { 255, 144,   0, 0 },  { 255, 146,   0, 0 },  { 255, 147,   0, 0 },  //164
{ 255, 149,   0, 0 },  { 255, 151,   0, 0 },  { 255, 153,   0, 0 },  { 255, 154,   0, 0 },  //168
{ 255, 156,   0, 0 },  { 255, 158,   0, 0 },  { 255, 160,   0, 0 },  { 255, 161,   0, 0 },  //172
{ 255, 163,   0, 0 },  { 255, 165,   0, 0 },  { 255, 167,   0, 0 },  { 255, 168,   0, 0 },  //176
{ 255, 170,   0, 0 },  { 255, 172,   0, 0 },  { 255, 174,   0, 0 },  { 255, 175,   0, 0 },  //180
{ 255, 177,   0, 0 },  { 255, 179,   0, 0 },  { 255, 181,   0, 0 },  { 255, 182,   0, 0 },  //184
{ 255, 184,   0, 0 },  { 255, 186,   0, 0 },  { 255, 188,   0, 0 },  { 255, 189,   0, 0 },  //188
{ 255, 191,   0, 0 },  { 255, 193,   0, 0 },  { 255, 195,   0, 0 },  { 255, 196,   0, 0 },  //192
{ 255, 198,   0, 0 },  { 255, 200,   0, 0 },  { 255, 202,   0, 0 },  { 255, 203,   0, 0 },  //196
{ 255, 205,   0, 0 },  { 255, 207,   0, 0 },  { 255, 209,   0, 0 },  { 255, 210,   0, 0 },  //200
{ 255, 212,   0, 0 },  { 255, 214,   0, 0 },  { 255, 216,   0, 0 },  { 255, 217,   0, 0 },  //204
{ 255, 219,   0, 0 },  { 255, 221,   0, 0 },  { 255, 223,   0, 0 },  { 255, 224,   0, 0 },  //208
{ 255, 226,   0, 0 },  { 255, 228,   0, 0 },  { 255, 230,   0, 0 },  { 255, 231,   0, 0 },  //212
{ 255, 233,   0, 0 },  { 255, 235,   0, 0 },  { 255, 237,   0, 0 },  { 255, 238,   6, 0 },  //216
{ 255, 240,  13, 0 },  { 255, 242,  19, 0 },  { 255, 244,  26, 0 },  { 255, 245,  32, 0 },  //220
{ 255, 247,  39, 0 },  { 255, 249,  45, 0 },  { 255, 251,  52, 0 },  { 255, 253,  58, 0 },  //224
{ 255, 255,  65, 0 },  { 255, 255,  71, 0 },  { 255, 255,  78, 0 },  { 255, 255,  84, 0 },  //228
{ 255, 255,  90, 0 },  { 255, 255,  96, 0 },  { 255, 255, 102, 0 },  { 255, 255, 108, 0 },  //232
{ 255, 255, 114, 0 },  { 255, 255, 120, 0 },  { 255, 255, 126, 0 },  { 255, 255, 132, 0 },  //236
{ 255, 255, 138, 0 },  { 255, 255, 144, 0 },  { 255, 255, 150, 0 },  { 255, 255, 156, 0 },  //240
{ 255, 255, 162, 0 },  { 255, 255, 168, 0 },  { 255, 255, 174, 0 },  { 255, 255, 182, 0 },  //244
{ 255, 255, 188, 0 },  { 255, 255, 194, 0 },  { 255, 255, 201, 0 },  { 255, 255, 207, 0 },  //248
{ 255, 255, 213, 0 },  { 255, 255, 219, 0 },  { 255, 255, 225, 0 },  { 255, 255, 231, 0 },  //252
{ 255, 255, 237, 0 },  { 255, 255, 243, 0 },  { 255, 255, 249, 0 },  { 255, 255, 255, 0 },  //256
};

// 编码表结束
};
```

CdlgColor 类是新添加的对话框类，该对话框的主要功能是让用户选择一个现有的伪彩色编码表。该对话框类的完整代码如下：

1. 对话框头文件 DlgColor.h：

```
#if !defined(AFX_DLGCOLOR_H__BC36E028_E94D_4AD1_AF17_E8F8E960D8F4__INCLUDED_)
#define AFX_DLGCOLOR_H__BC36E028_E94D_4AD1_AF17_E8F8E960D8F4__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgColor.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgColor dialog

class CDlgColor : public CDialog
{
// Construction
```

```
public:

    // 颜色名称字符串长度
    int m_nNameLen;

    // 颜色名称字符串数组指针
    LPSTR m_lpColorName;

    // 颜色数目
    int m_nColorCount;

    // 当前选择的伪彩色编码表
    int m_nColor;

    CDlgColor(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgColor)
    enum { IDD = IDD_DLG_Color };

    // ListBox控件对应的类成员变量
    CListBox    m_lstColor;

    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgColor)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgColor)
    afx_msg void OnDblclkColorList();
    virtual BOOL OnInitDialog();
    virtual void OnOK();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_DLGCOLOR_H__BC36E028_E94D_4AD1_AF17_E8F8E960D8F4__INCLUDED_)
```

2. 对话框实现代码 DlgColor.cpp:

```
// DlgColor.cpp : implementation file
//
#include "stdafx.h"
#include "ch1_1.h"
#include "DlgColor.h"

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgColor dialog

CDlgColor::CDlgColor(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgColor::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgColor)
        // NOTE: the ClassWizard will add member initialization here
    //}}AFX_DATA_INIT
}

void CDlgColor::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgColor)
    DDX_Control(pDX, IDC_COLOR_LIST, m_lstColor);
    //}}AFX_DATA_MAP
}

BEGIN_MESSAGE_MAP(CDlgColor, CDialog)
    //{{AFX_MSG_MAP(CDlgColor)
    ON_LBN_DBLCLK(IDC_COLOR_LIST, OnDblclkColorList)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgColor message handlers

BOOL CDlgColor::OnInitDialog()
{
    // 循环变量
    int     i;

    // 调用默认OnInitDialog函数
    CDialog::OnInitDialog();

    // 添加伪彩色编码
```

```
    for (i = 0; i < m_nColorCount; i++)
    {
        m_lstColor.AddString(m_lpColorName + i * m_nNameLen);
    }

    // 选中初始编码表
    m_lstColor.SetCurSel(m_nColor);

    // 返回TRUE
    return TRUE;

void CDlgColor::OnDblclkColorList()

    // 双击事件，直接调用OnOK()成员函数
    OnOK();

void CDlgColor::OnOK()

    // 用户单击确定按钮
    m_nColor = m_lstColor.GetCurSel();

    // 调用默认的OnOK()函数
    CDialog::OnOK();
```

图 6－19 为一幅医学图像（由核磁共振得到），利用上述代码对其进行伪彩色变换，采用彩虹编码 1 的结果如图 6－20 所示。

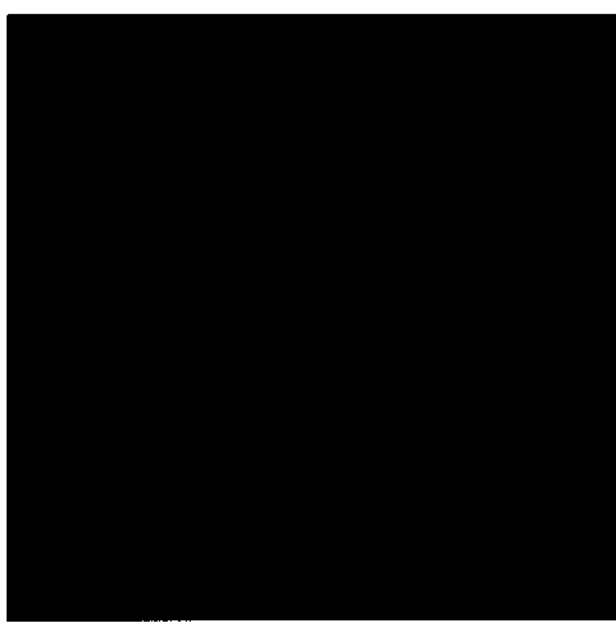


图 6－19　原始灰度图像

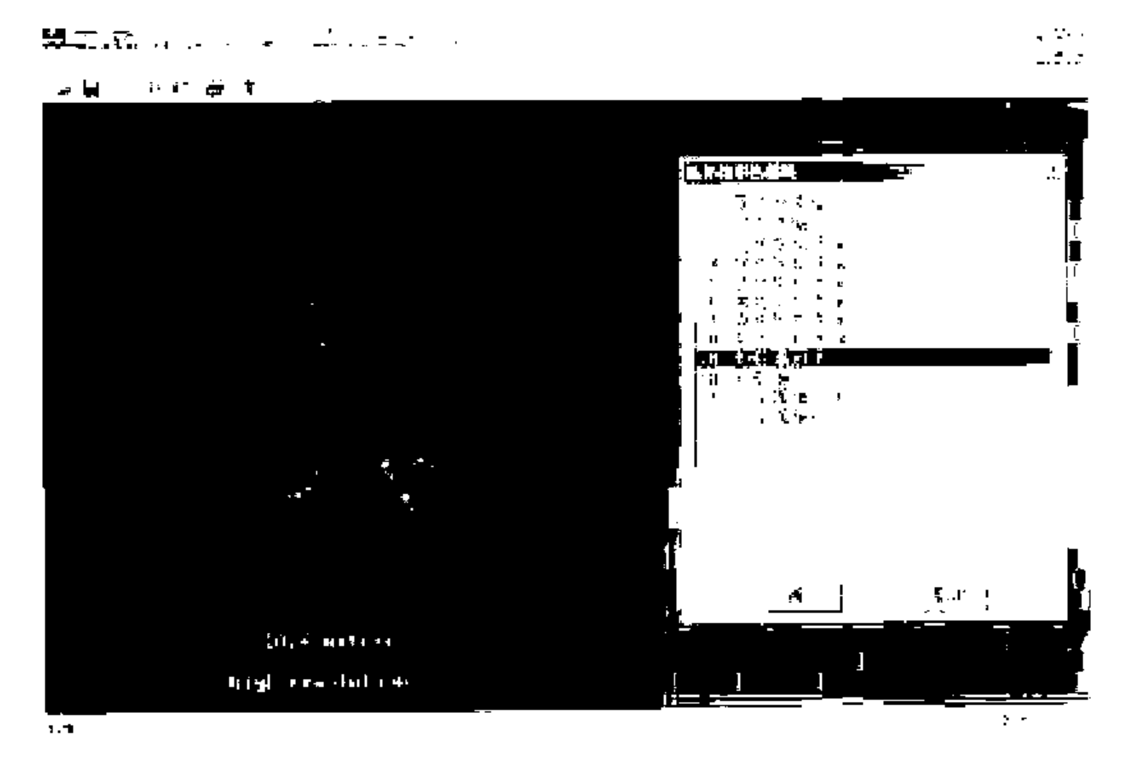


图 6－20 伪彩色变换后的结果

有了上面的伪彩色编码表，我们可以简单的编写一个将 256 色位图转换成灰度图的程序。

256 色位图转换成灰度图不能只是简单地用灰度调色板来替换 256 色调色板，否则，如图 6－21 所示的 256 色位图将变成如图 6－22 所示的黑白图像。



图 6－21 原始 256 色位图



图 6—22　用灰度调色板直接替换 256 色调色板结果

要将 256 色位图转换成灰度图，首先必须计算每种颜色对应的灰度值。在第二章中我们已经介绍了颜色系统，可以得到灰度和 RGB 颜色下面的对应关系：

$$Y = 0.299R + 0.587G + 0.114B$$

这样，按照上式我们可以方便地将 256 色调色板转换成灰度调色板。由于灰度图调色板一般是按照灰度逐渐上升循序排列的，因此我们还必须将图像每个像素值（即调色板颜色的索引值）进行调整。实际编程中只要定义一个颜色值到灰度值的映射表 bMap[256]（长为 256 的一维数组，保存 256 色调色板中各个颜色对应的灰度值），将每个像素值 p（即原 256 色调色板中颜色索引值）替换成 bMap[p]。

下面我们在文件菜单下添加一个名为“256 色位图－>灰度图”的菜单项（如图 6－23 所示），并在该菜单事件中添加如下代码以实现该操作。

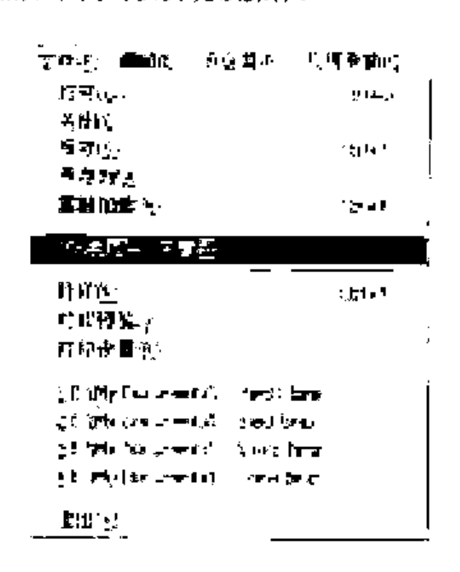


图 6—23　用灰度调色板直接替换 256 色调色板结果

```
void CCh1_1View::OnFILE256ToGray()
{
    // 将256色位图转换成灰度图

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 颜色表中的颜色数目
    WORD wNumColors;

    // 获取DIB中颜色表中的颜色数目
    wNumColors = ::DIBNumColors(lpDIB);

    // 判断是否是8-bpp位图
    if (wNumColors != 256)
    {
        // 提示用户
        MessageBox("非256色位图！", "系统提示" , MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 指向DIB像素的指针
    BYTE *  lpSrc;

    // 循环变量
    LONG    i;
    LONG    j;

    // 图像宽度
    LONG    lWidth;

    // 图像高度
    LONG    lHeight;
```

```
// 图像每行的字节数
LONG    lLineBytes;

// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFO lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREINFO lpbmc;

// 表明是否是Win3.0 DIB的标记
BOOL bWinStyleDIB;

// 获取指向BITMAPINFO结构的指针（Win3.0）
lpbmi = (LPBITMAPINFO)lpDIB;

// 获取指向BITMAPCOREINFO结构的指针
lpbmc = (LPBITMAPCOREINFO)lpDIB;

// 灰度映射表
BYTE bMap[256];

// 判断是否是WIN3.0的DIB
bWinStyleDIB = IS_WIN30_DIB(lpDIB);

// 计算灰度映射表（保存各个颜色的灰度值），并更新DIB调色板
for (i = 0; i < 256; i ++)
{
    if (bWinStyleDIB)
    {
        // 计算该颜色对应的灰度值
        bMap[i] = (BYTE)(0.299 * lpbmi->bmiColors[i].rgbRed +
                         0.587 * lpbmi->bmiColors[i].rgbGreen +
                         0.114 * lpbmi->bmiColors[i].rgbBlue + 0.5);

        // 更新DIB调色板红色分量
        lpbmi->bmiColors[i].rgbRed = i;

        // 更新DIB调色板绿色分量
        lpbmi->bmiColors[i].rgbGreen = i;

        // 更新DIB调色板蓝色分量
        lpbmi->bmiColors[i].rgbBlue = i;

        // 更新DIB调色板保留位
        lpbmi->bmiColors[i].rgbReserved = 0;
    }
    else
    {
        // 计算该颜色对应的灰度值
        bMap[i] = (BYTE)(0.299 * lpbmc->bmciColors[i].rgbtRed +
                         0.587 * lpbmc->bmciColors[i].rgbtGreen +
```

```
                    0.114 * lpbmc->bmciColors[i].rgbtBlue + 0.5);

            // 更新DIB调色板红色分量
            lpbmc->bmciColors[i].rgbtRed = i;

            // 更新DIB调色板绿色分量
            lpbmc->bmciColors[i].rgbtGreen = i;

            // 更新DIB调色板蓝色分量
            lpbmc->bmciColors[i].rgbtBlue = i;
        }
    }

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 获取图像宽度
    lWidth = ::DIBWidth(lpDIB);

    // 获取图像高度
    lHeight = ::DIBHeight(lpDIB);

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    // 更换每个像素的颜色索引（即按照灰度映射表换成灰度值）
    // 每行
    for(i = 0; i < lHeight; i++)
    {
        // 每列
        for(j = 0; j < lWidth; j++)
        {
            // 指向DIB第i行，第j个像素的指针
            lpSrc = (unsigned char*)lpDIBBits + lLineBytes * (lHeight - 1 - i) + j;

            // 变换
            *lpSrc = bMap[*lpSrc];
        }
    }

    // 替换当前调色板为灰度调色板
    pDoc->GetDocPalette()->SetPaletteEntries(0, 256, (LPPALETTEENTRY) ColorsTable[0]);

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 实现新的调色板
    OnDoRealize((WPARAM)m_hWnd,0);

    // 更新视图
    pDoc->UpdateAllViews(NULL);
```

```
	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();
}
```



图 6—24　256 色位图转换成灰度图

利用上述代码对图 6—21 转换的结果如图 6—24 所示。可以看出转换后的效果是令人满意的。

第七章 数字图像腐蚀、膨胀和细化算法

7.1 数学形态学

7.1.1 什么是数学形态学

数学形态学（Mathematical Morphology）是分析几何形状和结构的数学方法，是建立在集合代数基础上，用集合论方法定量描述几何结构的科学。1985年后，它逐渐成为分析图像几何特征的工具。腐蚀、膨胀和细化属于数学形态学范畴内的运算。

数学形态学是由一组形态学的代数运算子组成的。最基本的形态学运算子有：腐蚀（Erosion）、膨胀（Delation）、开（Opening）和闭（Closing）。用这些运算子及其组合来进行图像形状和结构的分析及处理，包括图像分割、特征抽取、边界检测、图像滤波、图像增强和恢复等方面的工作。

由于形态学具有完备的数学基础，这为形态学用于图像分析和处理、形态滤波器的特性分析和系统设计奠定了坚实的基础，尤其突出的是实现了形态学分析和处理算法的并行，大大提高了图像分析和处理的速度。近年来，在图像分析和处理中形态学的研究和应用在国外得到不断地发展。

数学形态学现在已经应用在多门学科的数字图像分析和处理的过程中。例如在医学和生物学中应用数学形态学对细胞进行检测、研究心脏的运动过程及对脊椎骨癌图像进行自动数量描述；在工业控制领域应用数学形态学进行食品检验（碎米）和电子线路特征分析；在交通管制中监测汽车的运动情况等等。另外，数学形态学在金相学、指纹检测、经济地理、合成音乐和断层X光照像等领域也有良好的应用前景。

形态学的理论基础是集合论。在图像处理中形态学的集合（Set）代表着黑白和灰度图像的形状，如黑白二值图像中所有黑色像素点（Pixel）的集合组成了此图像的完全描述。在一个集合中，将进行形态变换的像素点是被选择的集合 X，而此集合的补 X^c 是没有被选择的集合。通常被选择的集合是图像的前景（Foreground），而未被选择的集合是图像的背景（Background）。

7.1.2 数学形态学中的基本符号和术语

既然数学形态学是建立在集合论的基础之上的，那么在介绍数学形态学的算法之前，我们先来了解一些集合论和数学形态学中的符号和术语：

- 元素和集合

在数字图像处理的数学形态学运算中，我们把一幅图像称为一个集合。对于二值图像而言，习惯上认为取值为1的点对应于景物中心，而取值为0的点构成背景。这类图像的集合

是直接表示的。考虑所有1－值的点的集合（A），则A与图像是一一对应的。

对于一幅图像A，如果点a在A的区域以内，那么就说a是A的元素，记为$a \in A$，否则记作$a \notin A$，如图7－1所示。

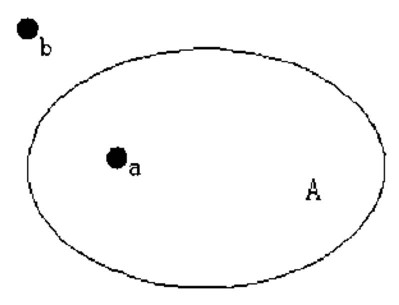


图7－1 $a \in A, b \notin A$

对于两幅图像A和B，如果对B中的每一个点b，$b \in B$都有$b \in A$，那么称B包含于A，记作$B \subseteq A$。如果同时还有A中存在至少一个点，$a \in A$且$a \notin B$，那么称B真包含于A，记作$B c A$，如图7－2所示。

根据定义可以知道，如果$B \subset A$，那么必有$B \subseteq A$；$A \subseteq A$恒成立。

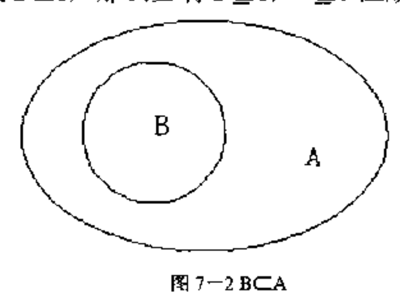


图7－2 $B \subset A$

- 交集、并集和补集

两个图像集合A和B的公共点组成的集合称为两个集合的交集，记为$A \cap B$，即

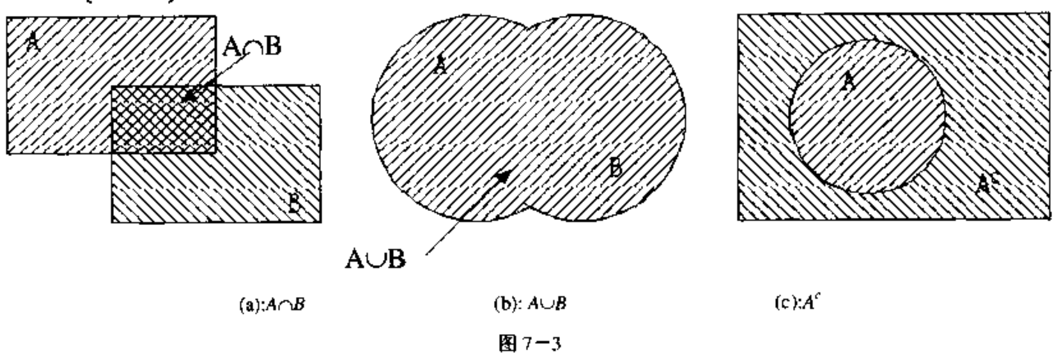
$A \cap B = \{a | a \in A$ 且 $a \in B\}$

两个集合A和B的公共元素组成的集合称为两个集合的并集，记为$A \cup B$，即

$A \cup B = \{a | a \in A$ 或 $a \in B\}$

对一幅图像A，在图像A区域以外的所有点构成的集合称为A的补集，记为A^c，即

$A^c = \{a | a \notin A\}$



(a):$A \cap B$　(b): $A \cup B$　(c):A^c

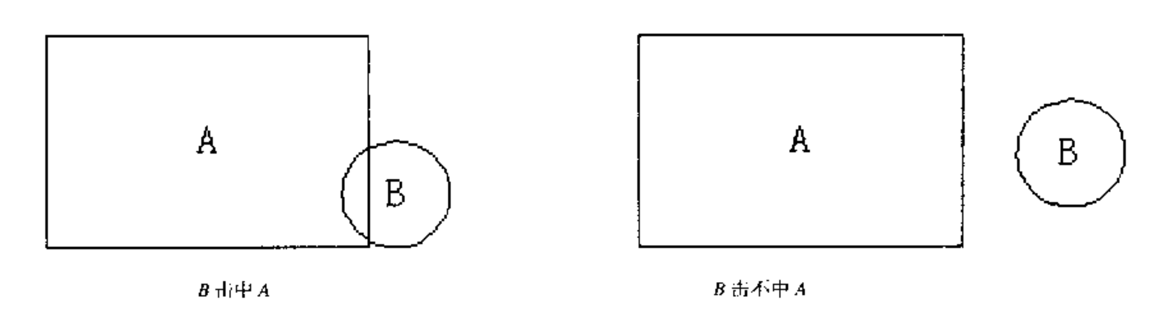
图7－3

交集、并集和补集运算是集合的最基本的运算。如图 7－3 所示。

● 击中(Hit)与击不中(Miss)

设有两幅图像 A 和 B，如果 $A\cap B\neq \Phi$，那么称 B 击中 A，记为 $B\uparrow A$ 其中 Φ是空集的符号。

否则如果 $A\cap B=\Phi$，那么称 B 击不中 A，如图 7－4 所示。



B 击中 *A*　　　　*B* 击不中 *A*

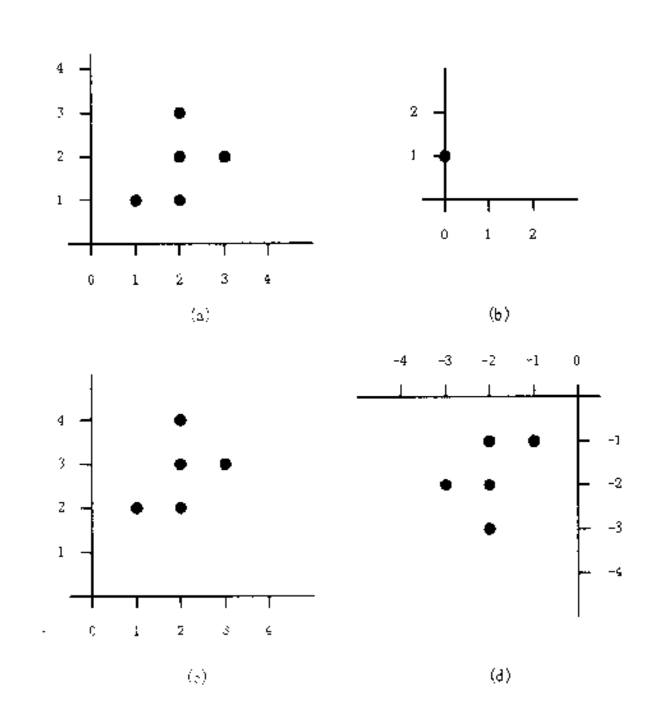
图 7－4

● 平移，反射

设 A 是一幅数字图像而 b 是一个点，那么定义 A 被 b 平移后的结果为

$A[b]=\{a+b|a\in A\}$

即取出 A 中的每个点 a 的坐标值，将其与 b 的坐标值相加，得到一个新的点的坐标值 $a+b$。所有这些新点所构成的图像就是 A 被 b 平移的结果，记为 $A[b]$。如图 7－5 所示。



(a) 图像 A （b）点 $b(0,1)$ （c）$A[b]$或 $A[0,1]$ （d）A^{-}

图 7－5 图像 A 被点 b 平移和被原点反射的结果

A 关于图像原点的反射结果为

$A^{\vee}=\{a|-a\in A\}$，

即将 A 中的每个点取相反数后所得的新图。

- 目标和结构元素

在后面的讨论中我们称被考察或者被处理的图像为目标图像，在本章中目标图像一般用 X 表示。

为了确定目标图像的结构，必须逐个地考察图像各部分之间的关系，并且进行检验。最后，得到一个各部分之间关系的集合。

在考察目标图像各部分之间的关系时，需要设计一种收集信息的“探针”，称为“结构元素”，在本章中一般用 S 表示。在图像中不断移动结构元素，就可以考察图像之间各部分的关系。

一般来说，结构元素的尺寸要明显小于目标图像的尺寸。

7.2 图像腐蚀（Erosion）

7.2.1 基本概念

数学形态学提出了一套独特的变换和运算方法。下面我们来看一看最基本的几种数学形态学运算。

对一个给定的目标图像 X 和一个结构元素 S，想像一下将 S 在图像上移动。在每一个当前位置 x，$S[x]$只有三种可能的状态，参见图 7－6：

(1) $S[x]\subseteq X$;

(2) $S[x]\subseteq X^c$;

(3) $S[x]\cap X$ 与 $S[x]\cap X^c$ 均不为空。

第一种情形说明 $S[x]$与 X 相关最大；第二种情形说明 $S[x]$与 X 不相关；而第三种情形说明 $S[x]$与 X 只是部分相关。因而满足（1）的点 x 的全体构成结构元素与图像的最大相关点集。我们称这个点集为 S 对 X 的腐蚀（简称腐蚀），记为 $X\ominus S$。也可以用集合的方式定义：

$X\ominus S=\{x\mid S[x]\subseteq X\}$

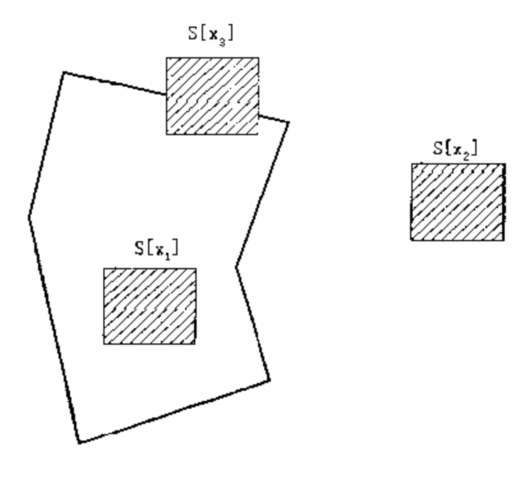


图 7－6 S[x]的三种可能状态

腐蚀在数学形态学运算中的作用是消除物体边界点。如果结构元素取 3×3 的黑点块，腐蚀将使物体的边界沿周边减少一个像素。

腐蚀可以把小于结构元素的物体去除，这样选取不同大小的结构元素，就可以去掉不同大小的物体。

如果两个物体之间有细小的连通，那么当结构元素足够大时，通过腐蚀运算可以将两个物体分开。

下面两张图是数学形态学中两个最基本的运算——腐蚀和膨胀的示意图。

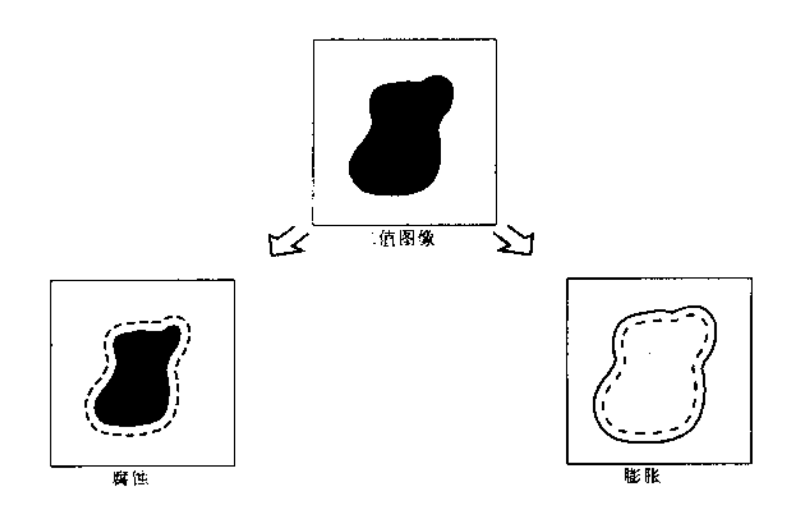


图 7－7 腐蚀和膨胀的示意图

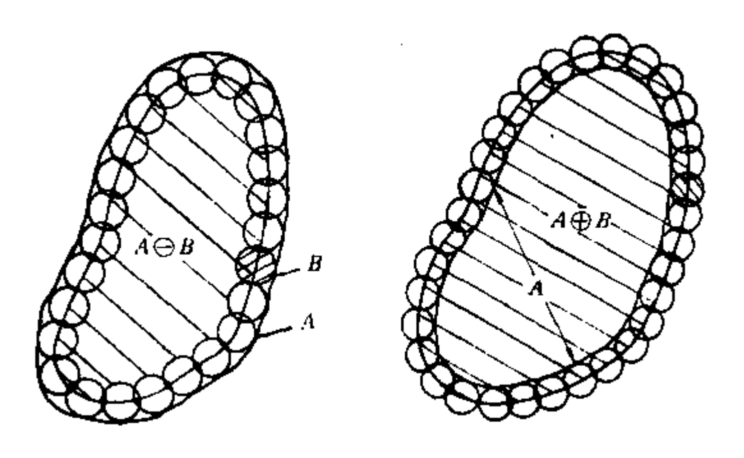
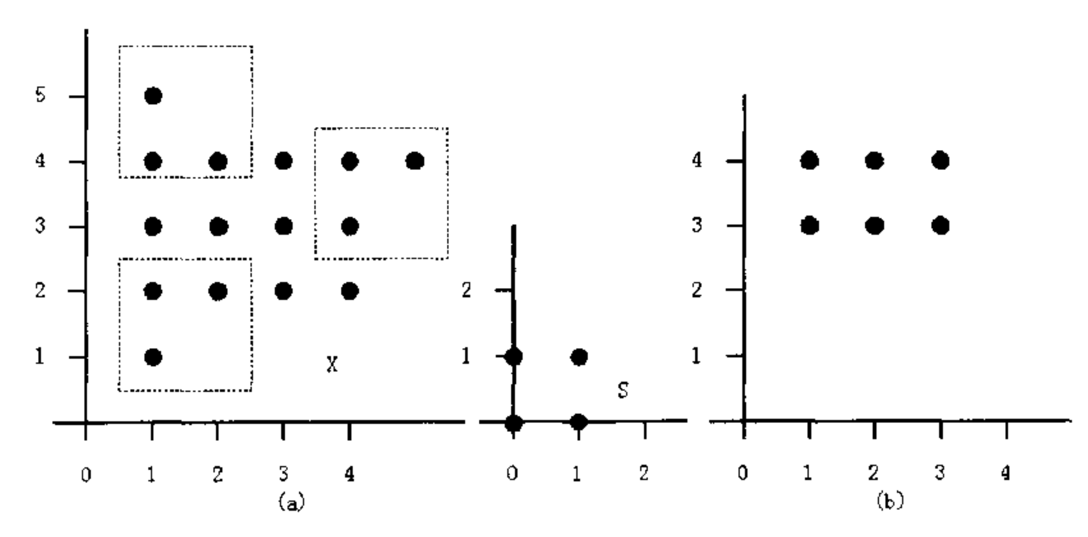


图 7—8 腐蚀和膨胀的示意图

下面我们来看看腐蚀运算的具体进行过程。

如图 7—9 所示，它相当于一个矩形加上了几个突出点。用图中的 S 对 X 进行腐蚀，由于在三个尖角处（图中用方框标出）都只有三点，不能与 S 重合，因此经腐蚀的图形消去了这些突出部分的点，同时剥去了 X 的上、右边界。



(a) 图像 X 和结构元素 (b) $X \ominus S$

图 7—9 X 被 S 腐蚀的几何解释

同样，对同一图像 X，结构元素不同时，腐蚀结果也不同。图 7—10 中画出了几种常见的情形。

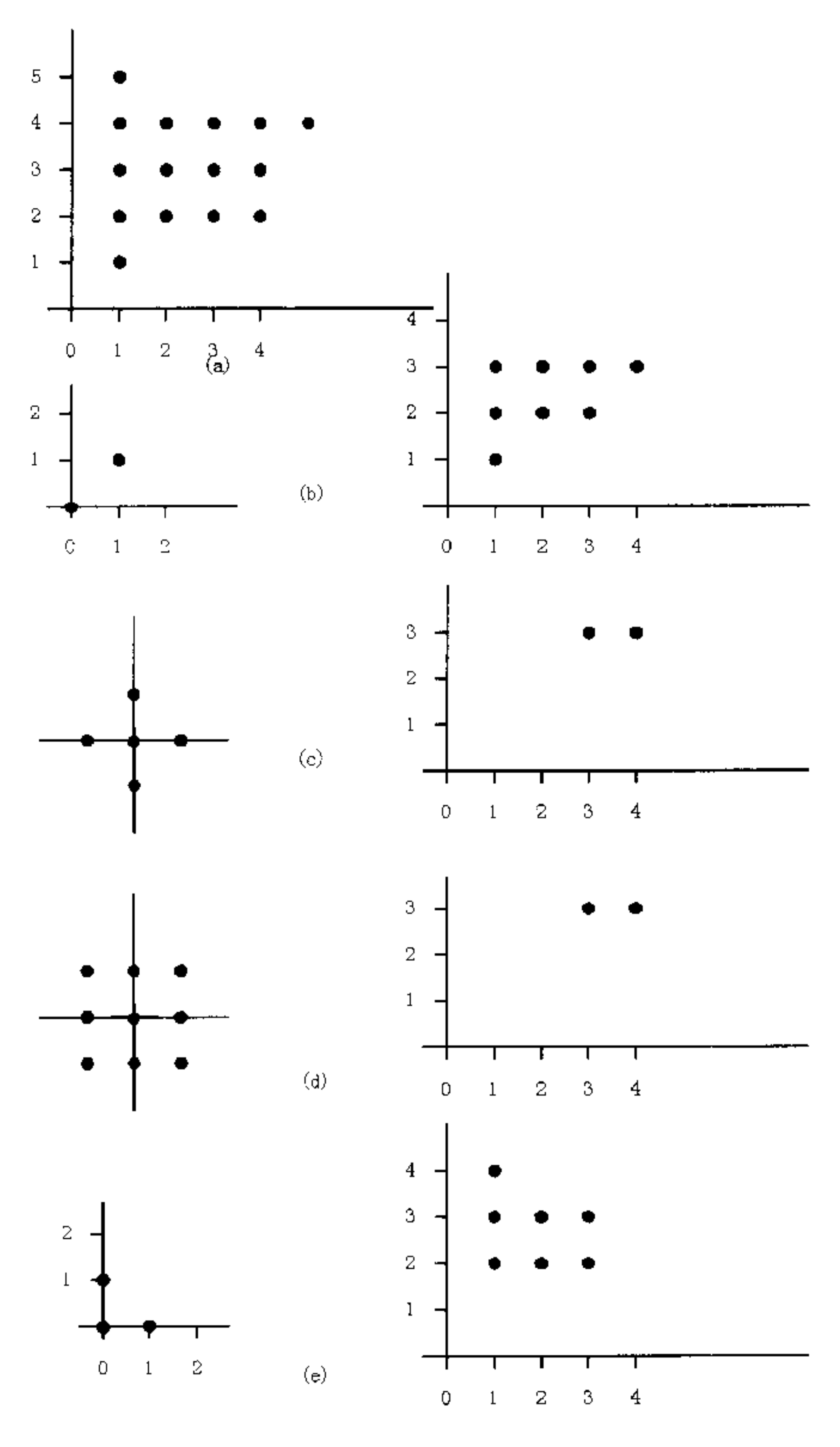


图 7-10 不同结构元素下的腐蚀结果 （a）图像 X （b）—（e）四种结构元素和相应的 $X \ominus S$

- 能否只把几个图像上的噪声点去掉而保持原图像 X 的尺寸不变呢？这时在经过腐蚀的图像上用同样的结构元素再进行一次膨胀运算，就可以达到这个目的了．关于膨胀运算的具体过程我们将在 1.3 节中详细介绍。对图像先进行一次腐蚀运算，再进行一次膨胀运算，在数学形态学上被称为"开运算"。在 1.4 节中将详细讨论开运算。
- 如果 S 包含了坐标原点 O，那么 $X\ominus S$ 将是 X 的一个收缩。即：
  $X\ominus S\subseteq X$（当 $O\in S$）
  但如果 S 不包含原点，那么 $X\ominus S\subseteq X$ 未必成立。如图 7－11 所示。
- 如果结构元素 S 关于原点 O 是对称的，那么 $S=S^v$，因此
  $X\ominus S=X\ominus S^v$（当 S 关于原点 O 对称）
  但如果 S 关于原点 O 不是对称的，那么 X 被 S 腐蚀的结果和 X 被 S^v 腐蚀的结果是不同的。如图 7－12 所示。

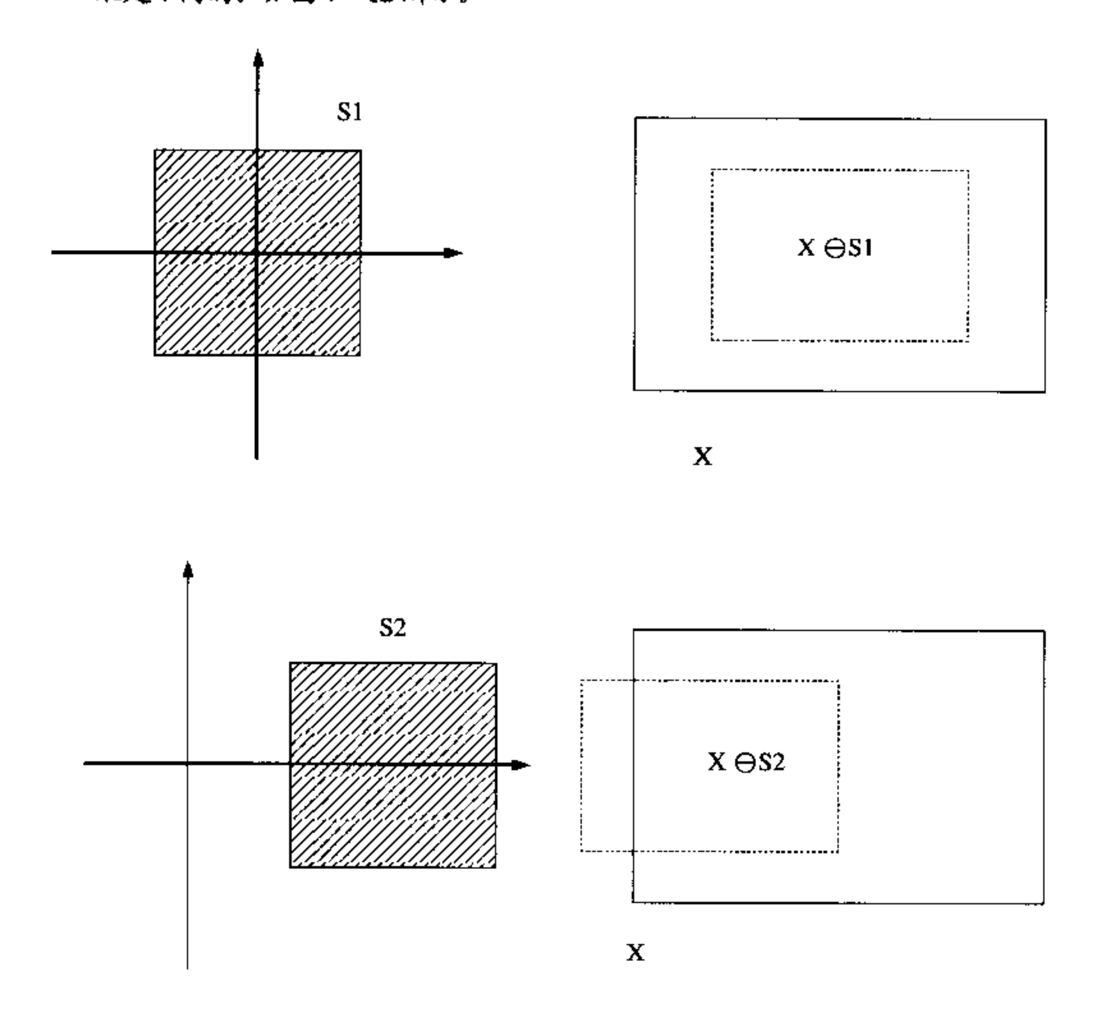


图 7－11 结构元素包含原点与否对腐蚀结果的影响

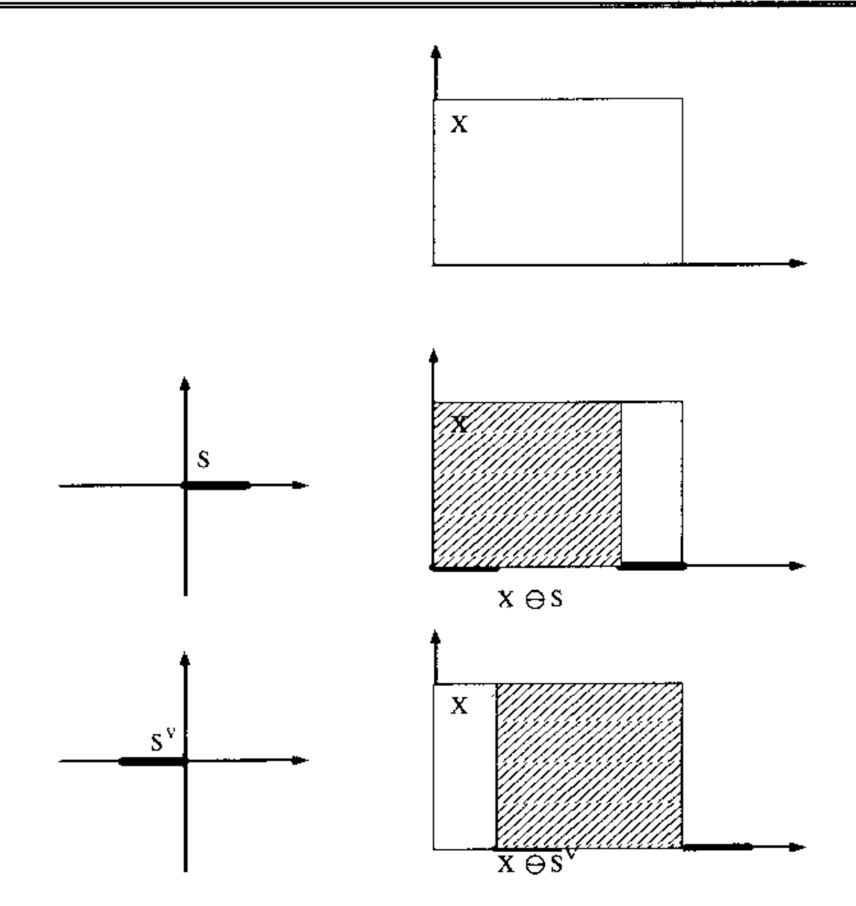


图 7－12 结构元素是否关于原点对称对腐蚀结果的影响

利用腐蚀运算的定义式可以直接设计腐蚀变换的算法。但有时，更为方便的是另一种表达式：

$X \ominus S = \cap \{ X[s] \mid -s \in S \}$

这一公式可从定义式中推出，它把腐蚀表示为图像平移的交，这在某些并行处理环境中特别有用。图 7－13 给出一个例子。

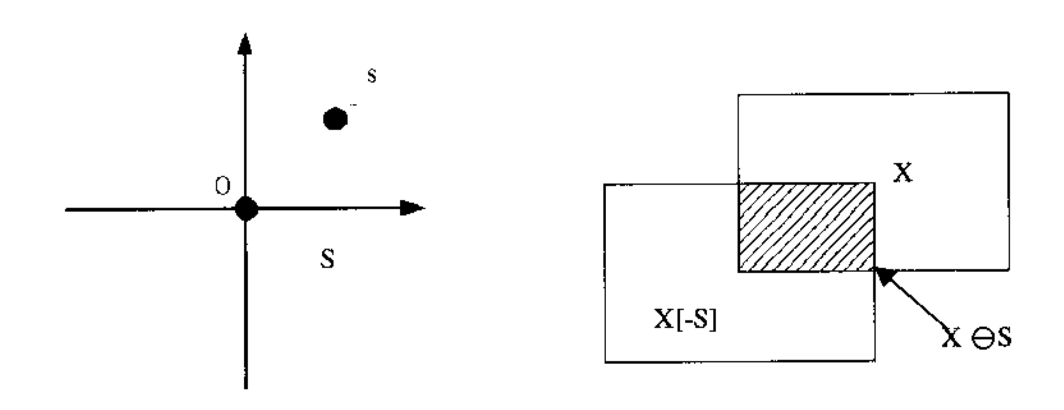


图 7－13 腐蚀表示为图像平移的交

7.2.2 Visual C++编程实现

下面我们来看一下数学形态学腐蚀运算的具体实现过程。

图像的腐蚀运算是由ErosionDIB()函数实现的。过程为：首先读入原图像中的一个点（称为当前点）的像素值，并把缓存图像中对应位置的点赋成黑色。然后把结构元素"覆盖"在原图像中以当前点为中心的区域上，如果在结构元素中某点为黑色而下面原图像中对应的点为白色，则把缓存图像中当前点赋成白色，即腐蚀掉。这样将原图像中所有的点读入一遍，在缓存图像中就得到了原图像的腐蚀结果。最后把缓存图像拷贝到原图像上并返回。

ErosionDIB()函数有几个参数：指向原 DIB 图像的指针，原图像的宽度和高度，腐蚀方式，结构元素。如果腐蚀方式 nMode 的值为 0，则使用{[-1,0]，[0,0]，[1,0]}的水平方向结构元素；如果腐蚀方式 nMode 的值为 1，则使用{[0,-1]，[0,0]，[0,1]}的垂直方向结构元素。如果腐蚀方式 nMode 的值为 2，则使用自定义的 3×3 结构元素。

因为我们的腐蚀运算是基于二值图像进行的，所以目前的 ErosionDIB()函数只支持 256 灰度图像，且图像中只能有 0 和 255 两种灰度值。

```
// ************************************************************************
//  文件名：morph.cpp
//
//  图像形态学变换API函数库：
//
//  ErosionDIB()     - 图像腐蚀
//  DilationDIB()  - 图像膨胀
//  OpenDIB()         - 图像开运算
//  CloseDIB()        - 图像闭运算
//  ThiningDIB()  - 图像细化
//
// ************************************************************************

#include "stdafx.h"
#include "morph.h"
#include "DIBAPI.h"

#include <math.h>
#include <direct.h>
/*************************************************************************
 *
 * 函数名称：
 *   ErosiontionDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG   lWidth          - 原图像宽度（像素数，必须是4的倍数）
 *   LONG   lHeight         - 原图像高度（像素数）
 *   int     nMode     - 腐蚀方式，0表示水平方向，1表示垂直方向，2表示自定义结构元素。
 *   int     structure[3][3]
 *                        - 自定义的3×3结构元素。
 *
 * 返回值:
 *   BOOL                    - 腐蚀成功返回TRUE，否则返回FALSE。
```

```
 *
 * 说明:
 * 该函数用于对图像进行腐蚀运算。结构元素为水平方向或垂直方向的三个点，中间点位于原点；
 * 或者由用户自己定义3×3的结构元素。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI ErosionDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, int nMode , int
structure[3][3])
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //循环变量
    long i;
    long j;
    int   n;
    int   m;

    //像素值
    unsigned char pixel;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);

    if(nMode == 0)
    {
        //使用水平方向的结构元素进行腐蚀
        for(j = 0; j <lHeight; j++)
```

```
		{
			for(i = 1;i <lWidth-1; i++)
			{
				//由于使用1×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素

				// 指向原图像倒数第i行，第j个像素的指针
				lpSrc = (char *)lpDIBBits + lWidth * j + i;

				// 指向目标图像倒数第i行，第j个像素的指针
				lpDst = (char *)lpNewDIBBits + lWidth * j + i;

				//取得当前指针处的像素值，注意要转换为unsigned char型
				pixel = (unsigned char)*lpSrc;

				//目标图像中含有0和255外的其他灰度值
				if(pixel != 255 && *lpSrc != 0)
					return FALSE;

				//目标图像中的当前点先赋成黑色
				*lpDst = (unsigned char)0;

				//如果原图像中当前点自身或者左右如果有一个点不是黑色，
				//则将目标图像中的当前点赋成白色
				for (n = 0;n < 3;n++ )
				{
					pixel = *(lpSrc+n-1);
					if (pixel == 255 )
					{
						*lpDst = (unsigned char)255;
						break;
					}
				}

			}
		}

	}
	else if(nMode == 1)
	{
		//使用垂直方向的结构元素进行腐蚀
		for(j = 1; j <lHeight-1; j++)
		{
			for(i = 0;i <lWidth; i++)
			{
				//由于使用1×3的结构元素，为防止越界，所以不处理最上边和最下边的两列像素

				// 指向原图像倒数第i行，第j个像素的指针
				lpSrc = (char *)lpDIBBits + lWidth * j + i;

				// 指向目标图像倒数第i行，第j个像素的指针
				lpDst = (char *)lpNewDIBBits + lWidth * j + i;
```

```
            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
            if(pixel != 255 && *lpSrc != 0)
                return FALSE;

            //目标图像中的当前点先赋成黑色
            *lpDst = (unsigned char)0;

            //如果原图像中当前点自身或者上下如果有一个点不是黑色，
            //则将目标图像中的当前点赋成白色
            for (n = 0;n < 3;n++ )
            {
                pixel = *(lpSrc+(n-1)*lWidth);
                if (pixel == 255 )
                {
                    *lpDst = (unsigned char)255;
                    break;
                }
            }

        }
    }

}
else
{
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            //由于使用3×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素
            //和最上边和最下边的两列像素
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
            if(pixel != 255 && *lpSrc != 0)
                return FALSE;

            //目标图像中的当前点先赋成黑色
            *lpDst = (unsigned char)0;
```

```
                //如果原图像中对应结构元素中为黑色的那些点中有一个不是黑色，
                //则将目标图像中的当前点赋成白色
                //注意在DIB图像中内容是上下倒置的
                for (m = 0;m < 3;m++ )
                {
                    for (n = 0;n < 3;n++)
                    {
                        if( structure[m][n] == -1)
                            continue;
                        pixel = *(lpSrc + ((2-m)-1)*lWidth + (n-1));
                        if (pixel == 255 )
                        {
                            *lpDst = (unsigned char)255;
                            break;
                        }
                    }
                }

            }
        }

    }
    // 复制腐蚀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

对应的头文件为：

```
// morph.h

#ifndef _INC_MorphAPI
#define _INC_MorphAPI

// 函数原型
BOOL WINAPI ErosionDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bHori , int structure[3][3]);
BOOL WINAPI DilationDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bHori , int structure[3][3]);
BOOL WINAPI OpenDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bHori , int structure[3][3]);
BOOL WINAPI CloseDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight, BOOL bHori , int structure[3][3]);
BOOL WINAPI ThiningDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);

#endif //!_INC_MorphAPI
```

下面添加一个数学形态学运算的菜单：

图 7－14 数学形态学变换菜单项

在该菜单的事件处理函数中，我们添加如下代码：

```
void CCh1_1View::OnMorphErosion()
{
    //腐蚀运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的腐蚀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的腐蚀！ ", "系统提示" , MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    int nMode;

    // 创建对话框
    cDlgMorphErosion dlgPara;

    // 初始化变量值
    dlgPara.m_nMode = 0;

    // 显示对话框，提示用户设定腐蚀方向
    if (dlgPara.DoModal() != IDOK)
```

```
	{
		// 返回
		return;
	}

	// 获取用户设定的腐蚀方向
	nMode = dlgPara.m_nMode;

	int structure[3][3];
	if (nMode == 2)
	{
		structure[0][0]=dlgPara.m_nStructure1;
		structure[0][1]=dlgPara.m_nStructure2;
		structure[0][2]=dlgPara.m_nStructure3;
		structure[1][0]=dlgPara.m_nStructure4;
		structure[1][1]=dlgPara.m_nStructure5;
		structure[1][2]=dlgPara.m_nStructure6;
		structure[2][0]=dlgPara.m_nStructure7;
		structure[2][1]=dlgPara.m_nStructure8;
		structure[2][2]=dlgPara.m_nStructure9;
	}

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用ErosionDIB()函数腐蚀DIB
	if (ErosionDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB), nMode ,
structure))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);

		// 更新视图
		pDoc->UpdateAllViews(NULL);
	}
	else
	{
		// 提示用户
		MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示" ,
MB_ICONINFORMATION | MB_OK);
	}

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
```

```
    // 恢复光标
    EndWaitCursor();

}
```

在资源中添加一个对话框，用于选择结构元素。利用classwizard为对话框中的控件加入成员变量，在对话框头文件cDlgMorphDilation.h中可以看到classwizard加入了下面的成员变量定义：

```
class cDlgMorphDilation : public CDialog
{
// Construction
public:
    cDlgMorphDilation(CWnd* pParent = NULL);    // standard constructor

// Dialog Data
    //{{AFX_DATA(cDlgMorphDilation)
    enum { IDD = IDD_DLG_MORPHDilation };
    CButton  m_Control9;
    CButton  m_Control8;
    CButton  m_Control7;
    CButton  m_Control6;
    CButton  m_Control5;
    CButton  m_Control4;
    CButton  m_Control3;
    CButton  m_Control2;
    CButton  m_Control1;
    int      m_nMode;
    int      m_nStructure1;
    int      m_nStructure2;
    int      m_nStructure3;
    int      m_nStructure4;
    int      m_nStructure5;
    int      m_nStructure6;
    int      m_nStructure7;
    int      m_nStructure8;
    int      m_nStructure9;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(cDlgMorphDilation)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(cDlgMorphDilation)
    afx_msg void OnHori();
```

```
    afx_msg void OnVert();
    afx_msg void Oncustom();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};
```

同时加入事件处理函数：

```
void cDlgMorphDilation::Oncustom()
{
    m_Control1.EnableWindow(TRUE);
    m_Control2.EnableWindow(TRUE);
    m_Control3.EnableWindow(TRUE);
    m_Control4.EnableWindow(TRUE);
    m_Control5.EnableWindow(TRUE);
    m_Control6.EnableWindow(TRUE);
    m_Control7.EnableWindow(TRUE);
    m_Control8.EnableWindow(TRUE);
    m_Control9.EnableWindow(TRUE);

}

void cDlgMorphDilation::OnVert()
{
    m_Control1.EnableWindow(FALSE);
    m_Control2.EnableWindow(FALSE);
    m_Control3.EnableWindow(FALSE);
    m_Control4.EnableWindow(FALSE);
    m_Control5.EnableWindow(FALSE);
    m_Control6.EnableWindow(FALSE);
    m_Control7.EnableWindow(FALSE);
    m_Control8.EnableWindow(FALSE);
    m_Control9.EnableWindow(FALSE);

}

void cDlgMorphDilation::OnHori()
{
    m_Control1.EnableWindow(FALSE);
    m_Control2.EnableWindow(FALSE);
    m_Control3.EnableWindow(FALSE);
    m_Control4.EnableWindow(FALSE);
    m_Control5.EnableWindow(FALSE);
    m_Control6.EnableWindow(FALSE);
    m_Control7.EnableWindow(FALSE);
    m_Control8.EnableWindow(FALSE);
    m_Control9.EnableWindow(FALSE);

}
```

这样当用户选择“自定义结构元素”的时候可以自行输入3×3的结构元素，而选择“水平方向”或者“垂直方向”结构元素的时候则禁止输入。

对话框、原图像及对原图像进行腐蚀的结果如图7－15、图7－16和图7－17所示。

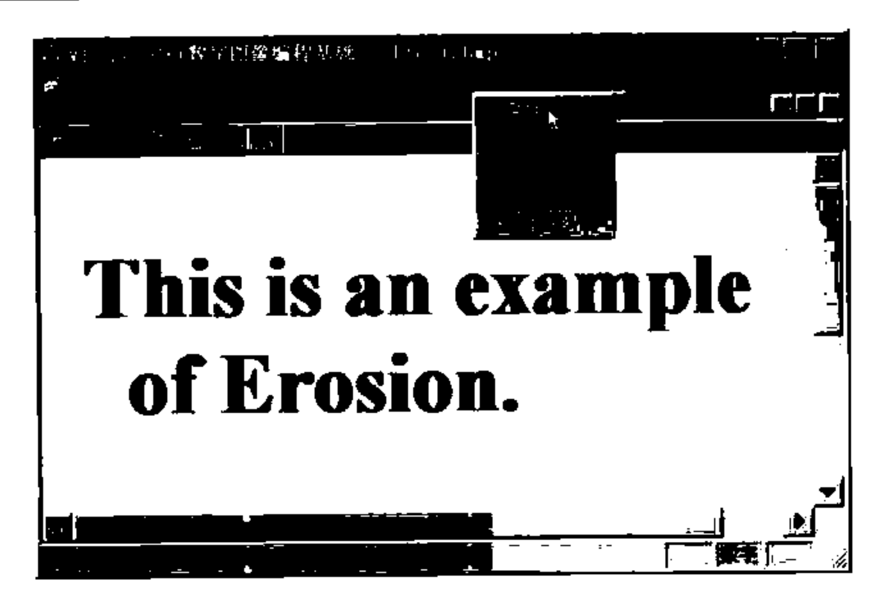


图7-15 原图像

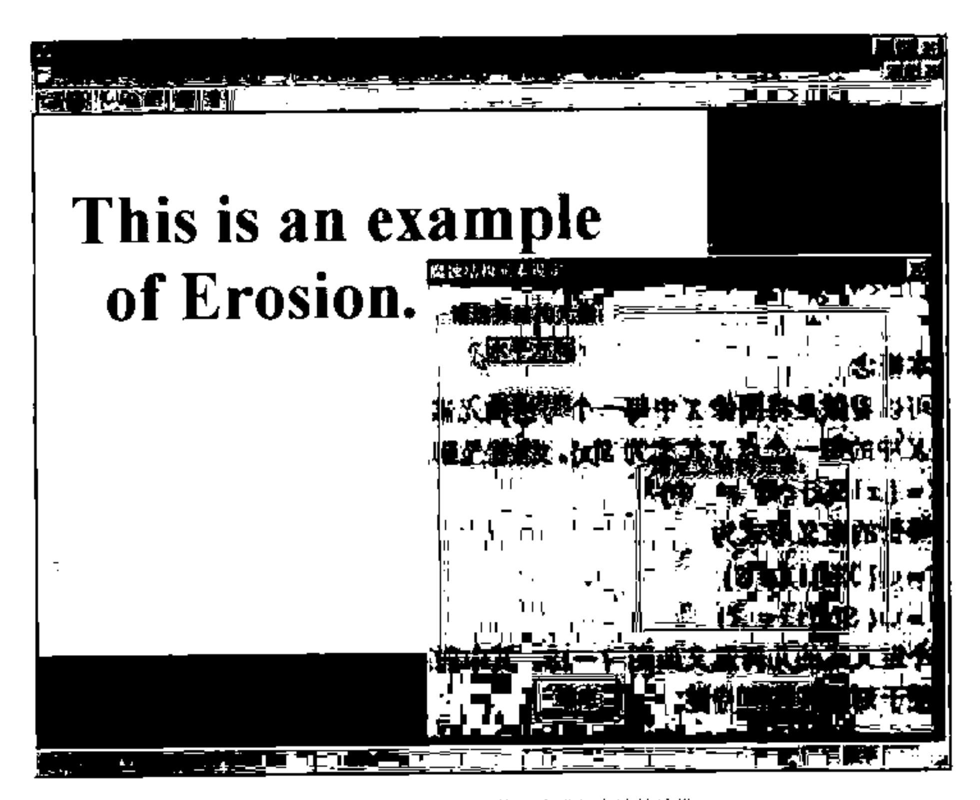


图7-16 用水平方向结构元素进行腐蚀的结果

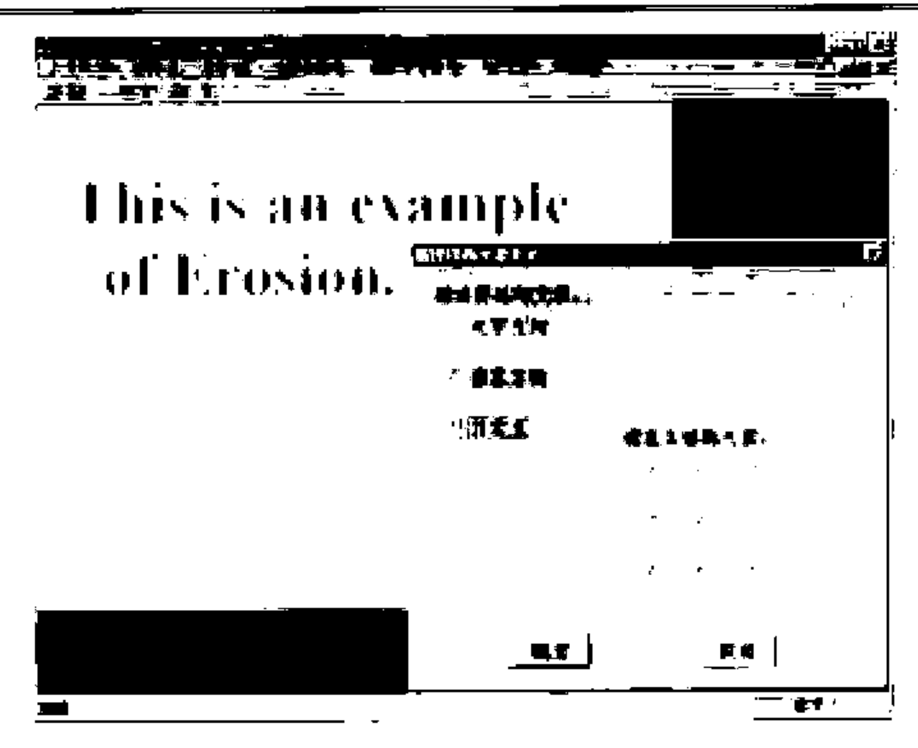


图 7－17 用九点结构元素进行腐蚀的结果

7.3 图像膨胀（Dilation）

7.3.1 基本概念

腐蚀可以看做是将图像 X 中每一个与结构元素 S 全等的子集 $S[x]$ 收缩为点 x。那么反之，也可以将 X 中的每一个点 X 扩大为 $S[x]$。这就是膨胀运算，记为 $X\oplus S$。它定义为

$X\oplus S=\{x \mid S[x]\cap x \neq \Phi\}$

与之等价的定义形式为

$X\oplus S=\cup\{X[s] \mid s\in S\}$

$X\oplus S=\cup\{S[x] \mid x\in X\}$

这三个定义式的几何意义如图 7－18。其中前两个式子在算法设计中更有用些，而后面一个式子便于刻画其几何特性。

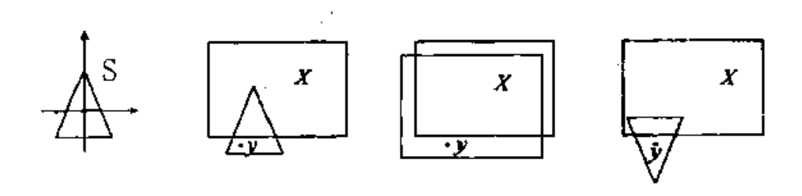


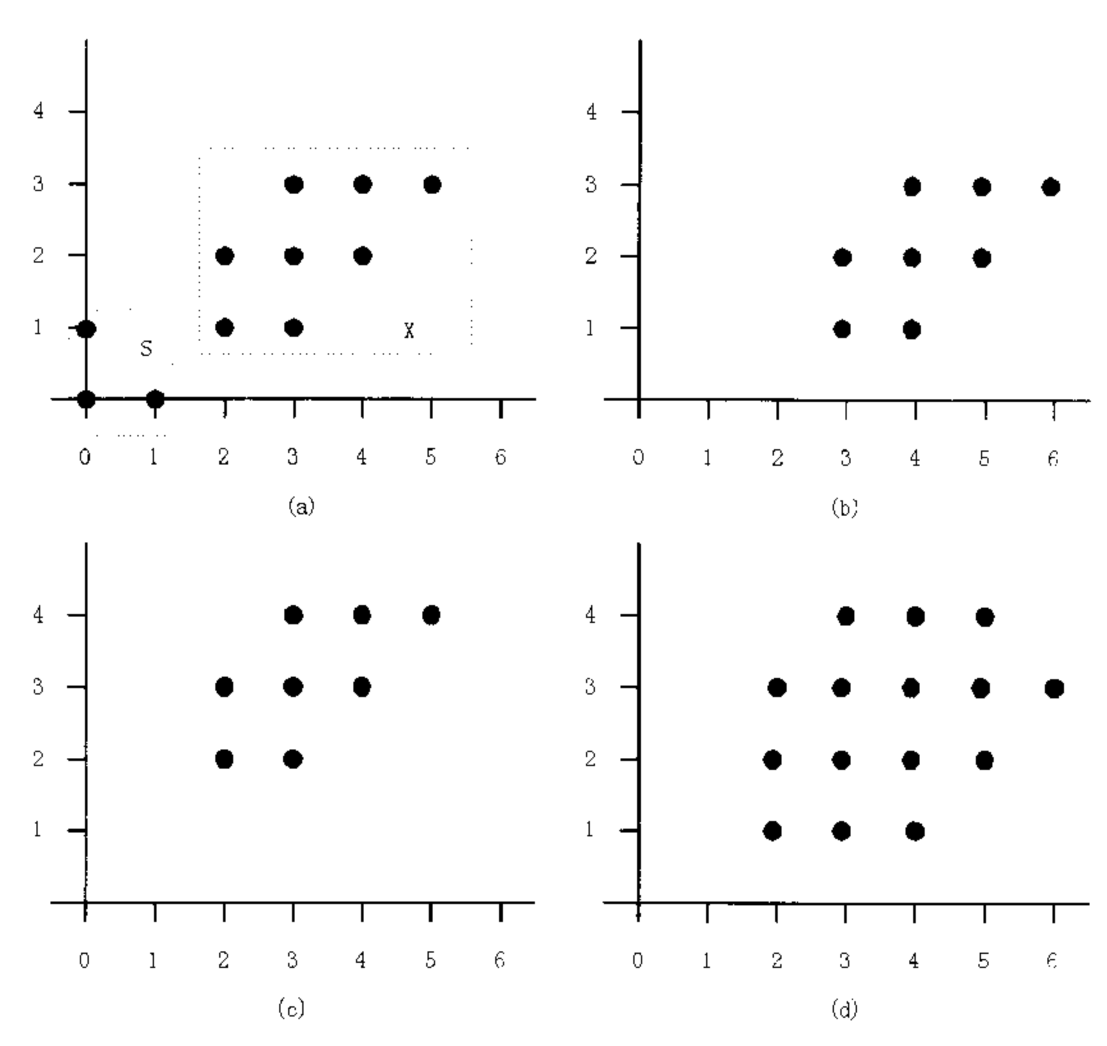
图 7－18 膨胀运算三种定义方式的图解，图中 $y\in X\oplus S$

下面来看一个根据第二个定义式进行膨胀运算的具体例子。

如图 7－19 所示。假设 S 中共包括三点，即 $S_1(0,0)$、$S_2(1,0)$ 和 $S_3(0,1)$。我们分别求 X 被 S_1、S_2 和 S_3 平移的结果，得到三个新图，其中 $X[S_1]$和 X 重合，$X[S_2]$相当于 X 向右平移一个单位而 $X[S_3]$相当于 X 向上平移一个单位。

下一步是将 $X[S_1]$，$X[S_2]$和 $X[S_3]$“合并”起来得到一幅新图像，也就是 X 被 S 膨胀的结果 $X \oplus S$。“合并”的含意是将 $X[S_1]$、$X[S_2]$和 $X[S_3]$重叠在一起，如果某个点(x,y)在以上三张图像中的灰度都是零，那么 $X \oplus S$ 在(x,y)处的灰度也取零；否则取 1。

由图 7－19 可见：X 被图中的结构元素 S 膨胀相当于在原有的 X 图像的基础上向右方和上方各扩充了一个单位，“膨胀”一词也就来源于此。



(a) 图像 X 和结构元素 S (b) $X[S_2]$ (c) $X[S_3]$ (d) $X \oplus S= X[S_1] \cup X[S_2] \cup X[S_3]$

图 7－19 图像 X 被结构元素 S 膨胀的结果

如果改变结构元素的形状，X 被 S 膨胀就会得到不同的结果。如图 7—20 所示：

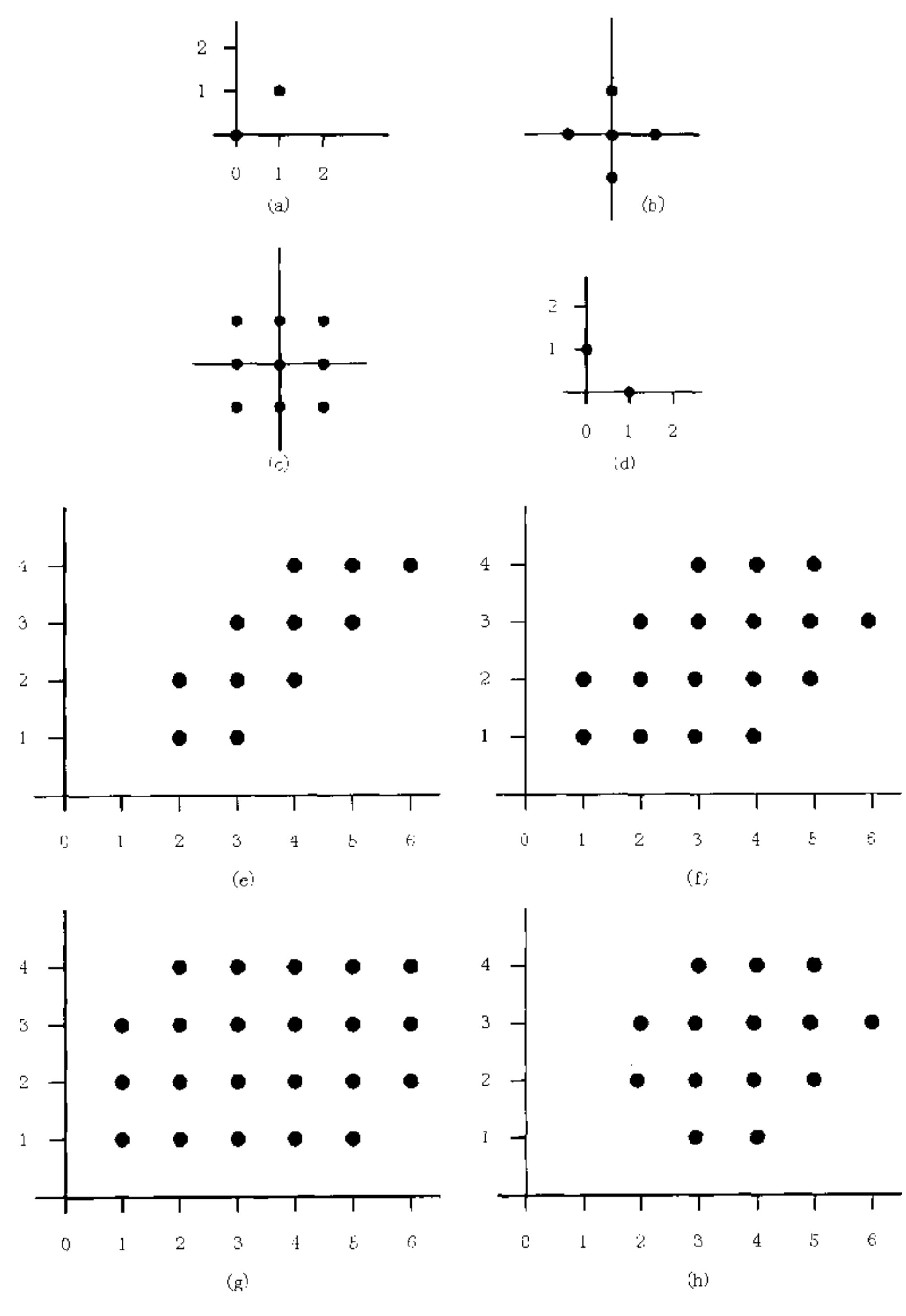


图 7—20 不同结构元素下的膨胀结果

图像 X 同图 7—19，(a)—(d) 为四种不同的结构元素 $S_1—S_4$，(e)—(h)为 X 在不同结构元素下的膨胀结果 $X \oplus S_1—X \oplus S_4$

> **如果结构元素 S 中不包含原点(0,0)，那么膨胀后的结果不一定包含 X 本身所有的点！即“膨胀”并不一定就是原图像的扩充。图 7－17(d)和(h)给出了一个这样的例子。**

膨胀运算在数学形态学中的作用是把图像周围的背景点合并到物体中。如果两个物体之间距离比较近，那么膨胀运算可能会使这两个物体连通在一起。膨胀对填补图像分割后物体中的空洞很有用。

7.3.2 腐蚀和膨胀的代数性质

当一个目标图像 X 在结构元素 S 之下进行了腐蚀或者膨胀运算后，根据所得的结果在位置或大小上的特点、和原来的图像 X 以及结构元素 S 之间的关系，下面将给出腐蚀和膨胀运算的代数性质。利用这些性质，也可以使运算过程得到一定程度的简化。

(1) 对偶性

$(X^c \ominus S)^c = X \oplus S$，$(X^c \oplus S)^c = X \ominus S$

腐蚀和膨胀运算的对偶性意味着腐蚀对应于补集的膨胀，反之亦然，这在理论和应用中都十分有用。此外，我们看到本质上形态学的基本变换只有一个，而整个形态学的体系均是建立在这一个变换之上的。

(2) 单调性

$X' \subseteq X \Rightarrow X' \ominus S \subseteq X \ominus S$，$X' \oplus S \subseteq X \ominus S$

$S' \subseteq S \Rightarrow X \ominus S' \supseteq X \ominus S$，$X \oplus S' \subseteq X \oplus S$

(3) 递（减）增性

$0 \in S \Rightarrow X \ominus S \subseteq X \subseteq X \oplus S$

实际上，在（2）的第二个式子里，如果令 $S'=\{(0,0)\}$，即 S 包含原点的话，就可以得到这一结果。

这一性质表明，在 S 包含原点的前提下，腐蚀会使 X 的点数减少或者不变，而膨胀则使 X 的点数增加或者不变。利用前一点，可以通过设计适当的结构元素 S，使得腐蚀后可以消除 X 中的微小颗粒即噪声点；利用后一点，又可以对腐蚀结果 $X \ominus S$ 再用 X 进行膨胀，以恢复有用信息。

(4) 交换律

$A \oplus B = B \oplus A$

注意：腐蚀运算没有交换性，即 $A \ominus B = B \ominus A$ 通常不成立。

(5) 结合律

$A \ominus (B \oplus C) = (A \ominus B) \ominus C$

$A \oplus (B \oplus C) = (A \oplus B) \oplus C$

这两个公式十分重要。它们表明采用一个较大结构元素 $B \oplus C$ 的形态学运算可以由两个采用较小结构元素 B 和 C 的形态学运算的级联来实现。这在实用中对增进算法的效率有很大的意义。由此所提出的结构元素分解问题我们将在第三章中详细讨论。

(6) 平移不变性

$X[h] \ominus S = (X \ominus S)[h]$

$X[h] \oplus S = (X \oplus S)[h]$

$(X \ominus S)[h] = (X \ominus S)[h]$

$(X \oplus S)[h] = (X \oplus S)[h]$

平移不变性意味着图像或结构元素的位置变化仅引起变换结果的位置变化，而结果集合的形态没有任何改变。这是数学形态学的创始人在设计数学形态学运算时提出的最基本的原则之一。同一个物体可能出现在图像的不同位置，对它们的分析不应因此而有所不同。

(7) 与集合运算的关系

$X \ominus (B \cup C) = (X \ominus B) \cap (X \ominus C)$

$X \oplus (B \cup C) = (X \oplus B) \cup (X \oplus C)$

$(X \cup Y) \ominus B \supseteq (X \ominus B) \cup (Y \ominus B)$

$(X \cup Y) \oplus B = (X \oplus B) \cup (Y \oplus B)$

$(X \cap Y) \ominus B = (X \ominus B) \cap (Y \ominus B)$

$(X \cap Y) \oplus B \subseteq (X \oplus B) \cap (Y \oplus B)$

我们看到腐蚀和膨胀运算对集合运算的分布律只有在特定情况下才能成立，所以在应用的时候要谨慎。

7.3.3 Visual C++编程实现

膨胀运算的过程由 DilationDIB()函数实现。膨胀运算和腐蚀运算的过程相类似，不同的是首先把缓存图像中当前点赋成白色，而只要结构元素中某点和原图像中与之对应的点都为黑色，则把缓存图像中当前点赋成黑色，即进行了膨胀。

DilationDIB()函数的参数和 ErosionDIB()是相同的：指向原 DIB 图像的指针，原图像的宽度和高度、腐蚀方式、结构元素。如果腐蚀方式 nMode 的值为 0，则使用{[-1,0]，[0,0]，[1,0]}的水平方向结构元素；如果腐蚀方式 nMode 的值为 1，则使用{[0,-1]，[0,0]，[0,1]}的垂直方向结构元素；如果腐蚀方式 nMode 的值为 2，则使用自定义的 3×3 结构元素。

和 ErosionDIB()函数一样，DilationDIB()函数只支持 256 灰度图像，且图像中只能有 0 和 255 两种灰度值。

```
/*************************************************************************
 *
 * 函数名称：
 *   DilationDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *   int   nMode          - 膨胀方式，0表示水平方向，1表示垂直方向，2表示自定义结构元素。
 *   int   structure[3][3]
 *                        - 自定义的3×3结构元素。
 *
 * 返回值:
 *   BOOL                 - 膨胀成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行膨胀运算。结构元素为水平方向或垂直方向的三个点，中间点位于原点；
 * 或者由用户自己定义3×3的结构元素。
```

```
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI DilationDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, int nMode , int
structure[3][3])
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //循环变量
    long i;
    long j;
    int   n;
    int   m;

    //像素值
    unsigned char pixel;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);


    if(nMode == 0)
    {
        //使用水平方向的结构元素进行膨胀
        for(j = 0; j <lHeight; j++)
        {
            for(i = 1;i <lWidth-1; i++)
            {
```

```
                //由于使用1×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素

                // 指向原图像倒数第j行，第j个像素的指针
                lpSrc = (char *)lpDIBBits + lWidth * j + i;

                // 指向目标图像倒数第j行，第j个像素的指针
                lpDst = (char *)lpNewDIBBits + lWidth * j + i;

                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                //目标图像中含有0和255外的其他灰度值
                if(pixel != 255 && *lpSrc != 0)
                    return FALSE;

                //目标图像中的当前点先赋成白色
                *lpDst = (unsigned char)255;

                //原图像中当前点自身或者左右只要有一个点是黑色，
                //则将目标图像中的当前点赋成黑色
                for (n = 0;n < 3;n++ )
                {
                    pixel = *(lpSrc+n-1);
                    if (pixel == 0 )
                    {
                        *lpDst = (unsigned char)0;
                        break;
                    }
                }

            }
        }

}
else if(nMode == 1)
{
    //使用垂直方向的结构元素进行膨胀
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            //由于使用1×3的结构元素，为防止越界，所以不处理最上边和最下边的两列像素

            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;
```

```
				//目标图像中含有0和255外的其他灰度值
				if(pixel != 255 && *lpSrc != 0)
					return FALSE;

				//目标图像中的当前点先赋成白色
				*lpDst = (unsigned char)255;

				//原图像中当前点自身或者上下只要有一个点是黑色,
				//则将目标图像中的当前点赋成黑色
				for (n = 0;n < 3;n++ )
				{
					pixel = *(lpSrc+(n-1)*lWidth);
					if (pixel == 0 )
					{
						*lpDst = (unsigned char)0;
						break;
					}
				}

			}
		}

}
else
{
	//使用自定义的结构元素进行膨胀
	for(j = 1; j <lHeight-1; j++)
	{
		for(i = 0;i <lWidth; i++)
		{
			//由于使用3×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素
			//和最上边和最下边的两列像素
			// 指向原图像倒数第j行，第i个像素的指针
			lpSrc = (char *)lpDIBBits + lWidth * j + i;

			// 指向目标图像倒数第j行，第i个像素的指针
			lpDst = (char *)lpNewDIBBits + lWidth * j + i;

			//取得当前指针处的像素值，注意要转换为unsigned char型
			pixel = (unsigned char)*lpSrc;

			//目标图像中含有0和255外的其他灰度值
			if(pixel != 255 && *lpSrc != 0)
			{
				return FALSE;
			}

			//目标图像中的当前点先赋成白色
			*lpDst = (unsigned char)255;
```

```
                //原图像中对应结构元素中为黑色的那些点中只要有一个是黑色，
                //则将目标图像中的当前点赋成黑色
                //注意在DIB图像中内容是上下倒置的
                for (m = 0;m < 3;m++ )
                {
                    for (n = 0;n < 3;n++)
                    {
                        if( structure[m][n] == -1)
                            continue;
                        pixel = *(lpSrc + ((2-m)-1)*lWidth + (n-1));
                        if (pixel == 0 )
                        {
                            *lpDst = (unsigned char)0;
                            break;
                        }
                    }
                }

            }
        }

    }
    // 复制膨胀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

在ch1_1view中加入下面的事件处理代码:

```
void CCh1_1View::OnMorphDilation()
{
    //膨胀运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
```

```
	{
		// 提示用户
		MessageBox("目前只支持256色位图的膨胀！", "系统提示", MB_ICONINFORMATION |
MB_OK);

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 返回
		return;
	}

	int nMode;

	// 创建对话框
	cDlgMorphDilation dlgPara;

	// 初始化变量值
	dlgPara.m_nMode = 0;

	// 显示对话框，提示用户设定膨胀方向
	if (dlgPara.DoModal() != IDOK)
	{
		// 返回
		return;
	}

	// 获取用户设定的膨胀方向
	nMode = dlgPara.m_nMode;

	int structure[3][3];
	if (nMode == 2)
	{
		structure[0][0]=dlgPara.m_nStructure1;
		structure[0][1]=dlgPara.m_nStructure2;
		structure[0][2]=dlgPara.m_nStructure3;
		structure[1][0]=dlgPara.m_nStructure4;
		structure[1][1]=dlgPara.m_nStructure5;
		structure[1][2]=dlgPara.m_nStructure6;
		structure[2][0]=dlgPara.m_nStructure7;
		structure[2][1]=dlgPara.m_nStructure8;
		structure[2][2]=dlgPara.m_nStructure9;
	}

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
```

```
        lpDIBBits = ::FindDIBBits(lpDIB);

        // 调用DilationDIB()函数膨胀DIB
        if (DilationDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB), nMode ,
structure))
        {

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示",
MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();
    }
```

膨胀运算的结果如下图所示。

This is the
example of Dilation.

图 7－21 原图像

This is the
example of Dilation.

图 7－22 用水平方向结构元素进行膨胀的结果

This is the example of Dilation.

图 7－23 用九点结构元素进行膨胀的结果

> 用腐蚀和膨胀运算还可以实现图像的平移。如果在自定义结构元素的时候选择不在原点的一个点作为结构元素，则得到的图像形状没有任何改变，只是位置发生了移动。

7.4 开运算（Open）和闭运算（Close）

7.4.1 基本概念

在腐蚀和膨胀两个基本运算的基础上，我们可以构造出形态学运算族，它由上述两个运算的复合和集合操作（并、交、补等）组合成的所有运算构成。其中两个最为重要的组合运算是形态学开运算和闭运算。

对图像 X 及结构元素 S，用 $X \circ S$ 表示 X 对 S 的开运算，用 $X \bullet S$ 表示 X 对 S 的闭运算，则它们的定义为：

$X \circ S = (X \ominus S) \oplus S$

$X \circ S = (X \oplus S) \ominus S$

因此，$X \circ S$ 可视为对腐蚀图像 $X \ominus S$ 用膨胀来进行恢复。而 $X \bullet S$ 可看作是对膨胀图像 $X \oplus S$ 用腐蚀来进行恢复。不过这一恢复不是信息无损的，即它们通常不等于原始图像 X。由开运算的定义式，我们可以推得

$X \circ S = \cup\{ S[x] \mid S[x] \in X\}$

因而 $X \circ S$ 是所有 X 的与结构元素 S 全等的子集的并组成的。或者说对 $X \circ S$ 中的每一个点 X，我们均可找到某个包含在 X 中的结构元 S 的平移 $S[y]$，使得 $X \in S[y]$，即 X 在 X 的近旁具有不小于 S 的几何结构。而对于 X 中不能被 $X \circ S$ 恢复的点，其近旁的几何结构总比 S 要小。这一几何描述说明 $X \circ S$ 是一个基于几何结构的滤波器。图 7－24 给出了两个开运算的例子。当使用圆盘结构元素时，开运算对边界进行了平滑，去掉了凸角。在凸角点周围，图像的几何构形无法容纳给定的圆盘，从而使凸角点周围的点被开运算删除。而当使用线段结构元素时，沿线段方向宽度较大的部分才能够被保留下来，而较小的凸部分将被剔除。因此 $X \circ S$ 及 $X - X \circ S$ 恰好形成了 X 的分割且分别包含了 X 的具有不同几何结构的部分。前者给出与结构元素相匹配的部分，而后者给出不相匹配的部分。不同的结构元素的选择导致了不同的分割，即提取出不同的特征。

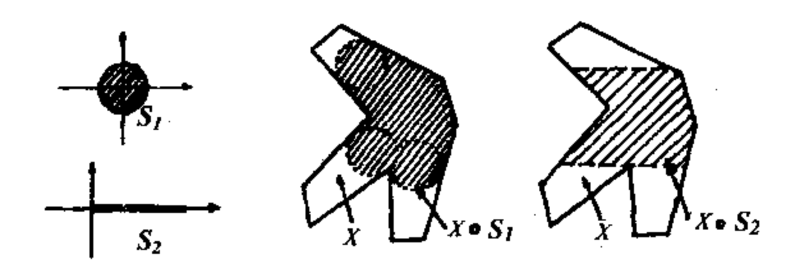


图 7－24 开变换图例。注意它们去掉了凸角

由腐蚀和膨胀的对偶性，可知

$(X^c \circ S)^c = X \bullet S$

$(X^c \bullet S)^c = X \circ S$

开、闭变换也是一对对偶变换。所以，闭运算的几何意义可以由补集的开运算的几何意义导出。图 7－25 给出了两个闭运算的例子。可见闭运算通过填充图像的凹角点来平滑图像，而 $X \bullet S - X$ 给出的是图像的凹入特征。

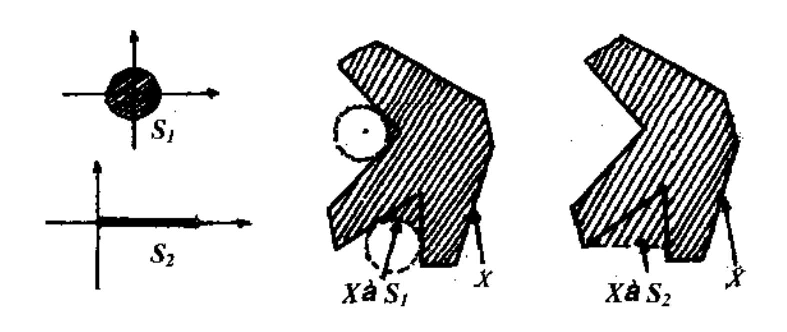
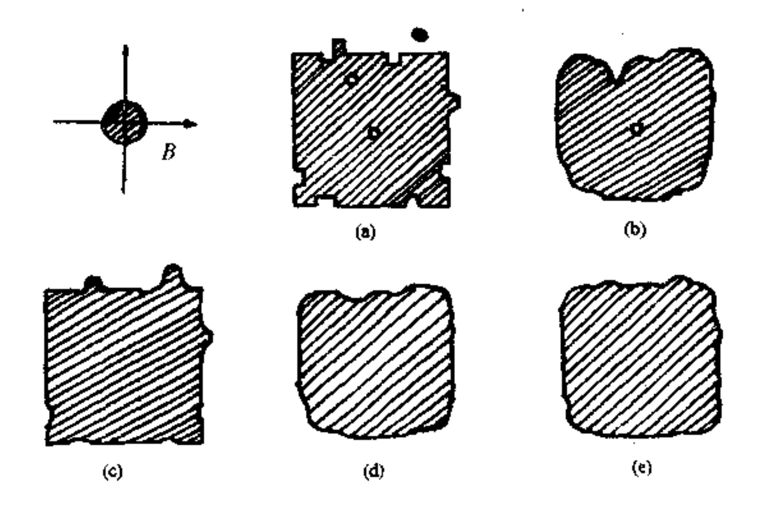


图 7－25 开变换图例。注意它们填充了凹角

由于开、闭变换所处理的信息分别与图像的凸、凹处相关，因此，它们均是单边算子。为此，人们提出了级联滤波器，即采取开、闭运算的交替使用来达到双边滤波的目的，例如（$X \circ S$）$\bullet S$，或（$X \bullet S$）$\circ S$ 等。

在图像处理的过程中，可以利用开、闭运算来去除噪声，恢复图像。在图 7－19 中描述了一个简单的示例。其中（a）是一个加了噪声的正方形物体的图像 X。噪声在背景中产生了小的斑块，在物体内部产生了洞，在物体边缘产生了锯齿状变形。（b）、（c）、（d）、（e）分别示出了 $X \circ S$、$X \bullet S$、（$X \circ S$）$\bullet S$、和（$X \bullet S$）$\circ S$。这里 S 采用的是圆盘结构元素，这样可以保证算子对图像的旋转是不变的。结构元素的直径应比噪声信号直径略大。这样一来，斑块噪声和洞不可能包含这一圆盘，从而分别为 $X \circ S$ 和 $X \bullet S$ 所去除，边界上的凸、凹变形也分别被 $X \circ S$ 和 $X \bullet S$ 所平滑。从运算的结果中可以看出 $X \circ S$，$X \bullet S$ 的单边滤波特点以及（$X \circ S$）$\bullet S$ 和（$X \bullet S$）$\circ S$ 的双边滤波性质。



（a）噪声图像 X；（b）$X \circ S$；（c）$X \bullet S$；（d）（$X \circ S$）$\bullet S$；（e）（$X \bullet S$）$\circ S$

图 7－26 形态学单边和双边滤波

从图 7－26（d）和（e）可以看到，（$X \bullet S$）$\circ S$ 和（$X \circ S$）$\bullet S$ 的效果是不同的。在本示例中，（$X \bullet S$）$\circ S$ 效果更好。在实际的应用中，哪一个性能较好则取决于噪声信号的性质。

下面来看几个具体实现开运算和闭运算的例子。

如图 7－27 所示，利用开运算可以消除散点和“毛刺”，也就是对图像进行平滑。

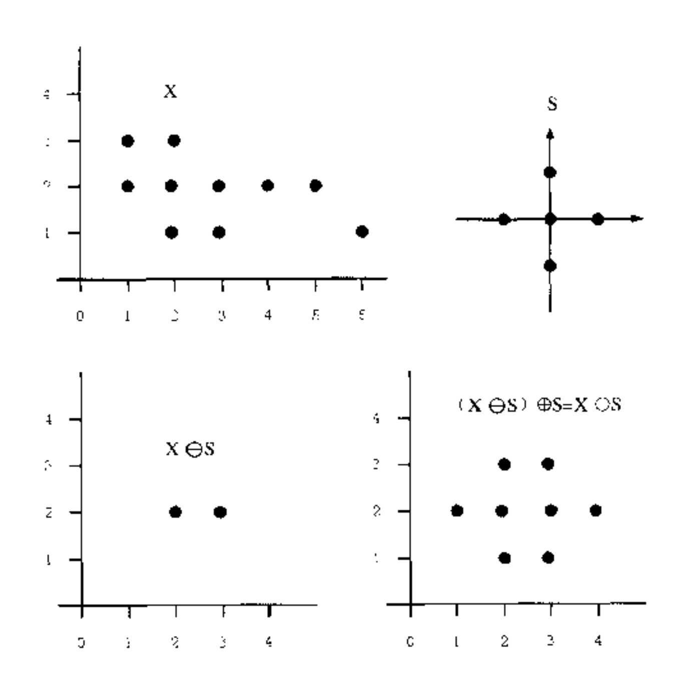


图 7－27 开运算示意图

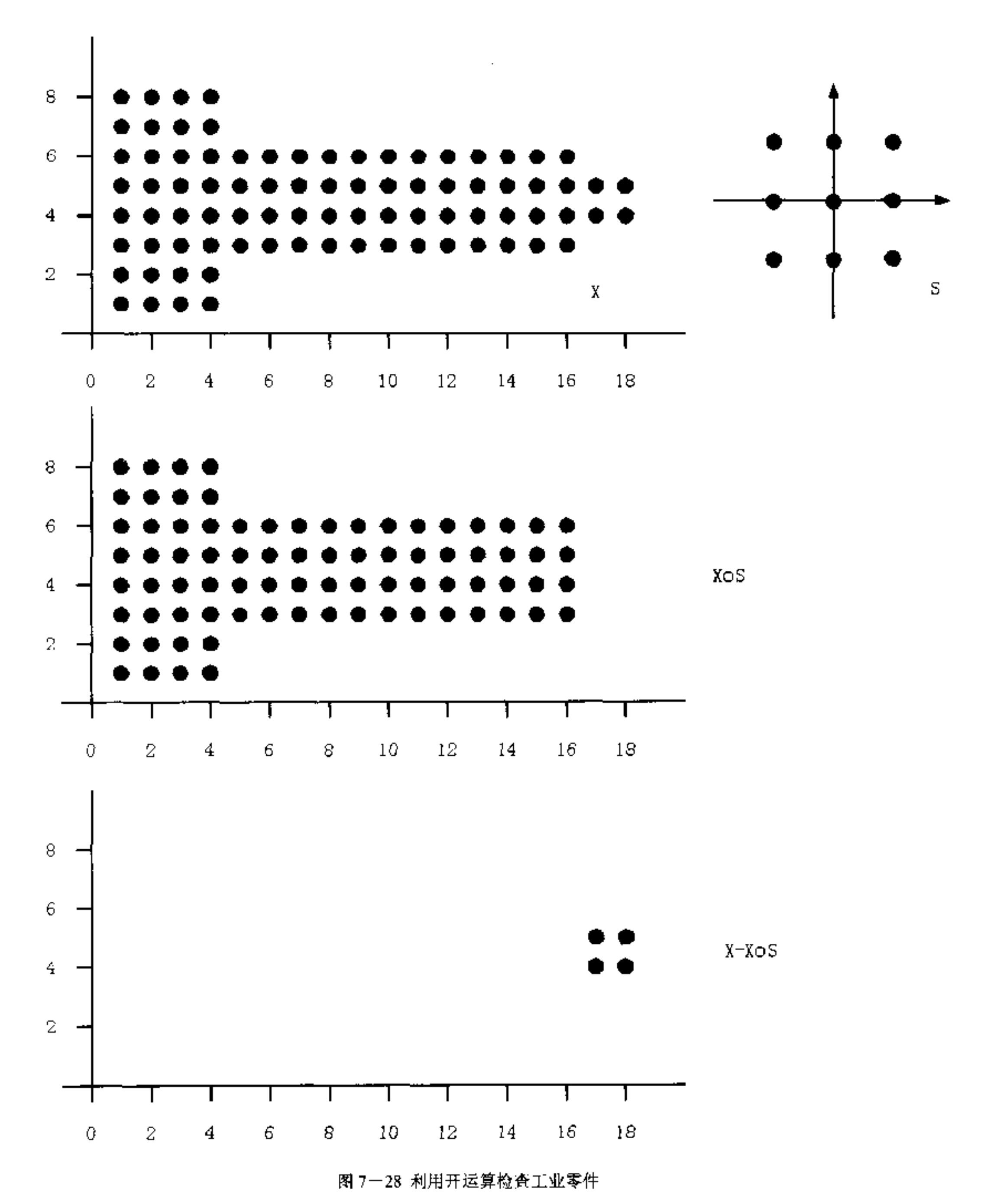


图 7－28 利用开运算检查工业零件

图 7－28 举出了利用开运算检查零件形状的例子。从 X 中减去 $X \circ S$ 便得到 X 的“毛刺”部分。对于图中画出的标准零件，可以通过检验毛刺部分的大小来判断它是否合格。

图 7－29 画出了一个闭运算的例子。由这个例子可以看出，在选择了适当的结构元素后，

可以通过闭运算将两个邻近的目标连接起来。

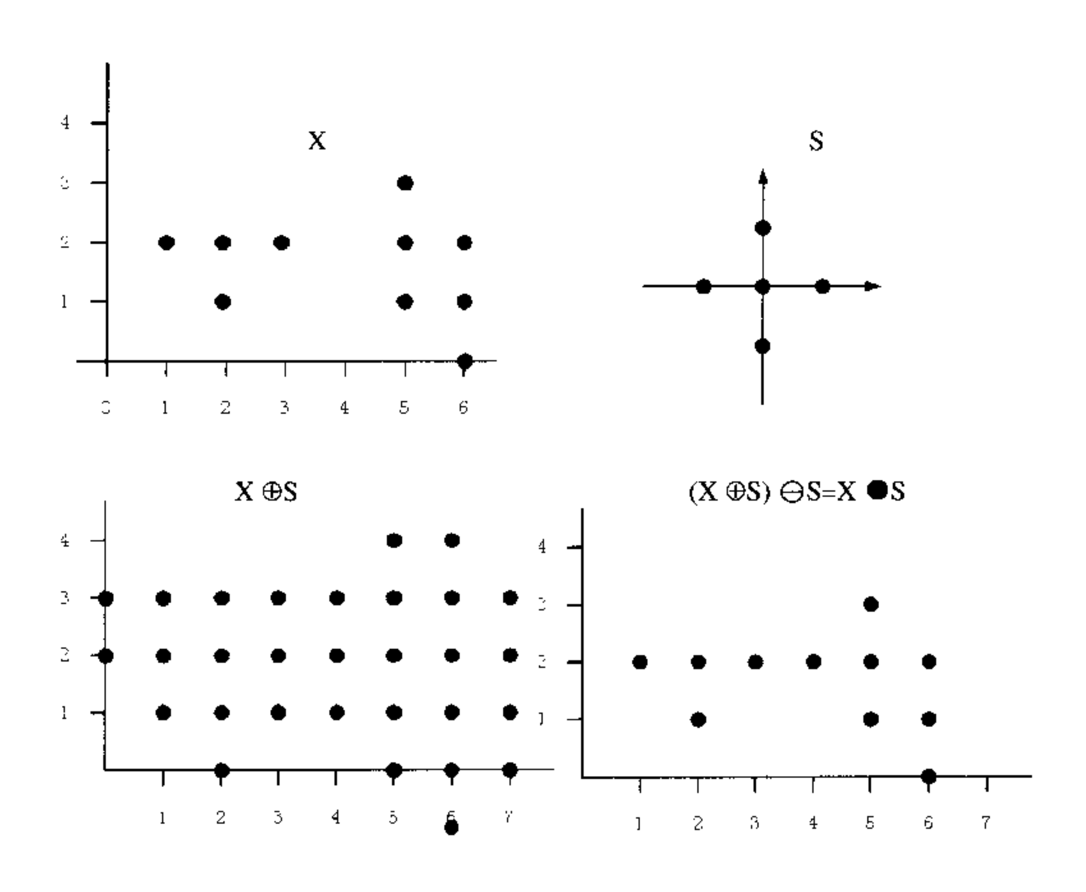


图 7－29 闭运算示意图

图 7－30 画出了另一个例子。选择图中所示的结构元素 S，它包括原点和位于第一象限中的三个点。从 X 被 S 执行膨胀运算的结果中去掉 X●S，就得到了 S 在“东北方向”上的一条外部轮廓线。

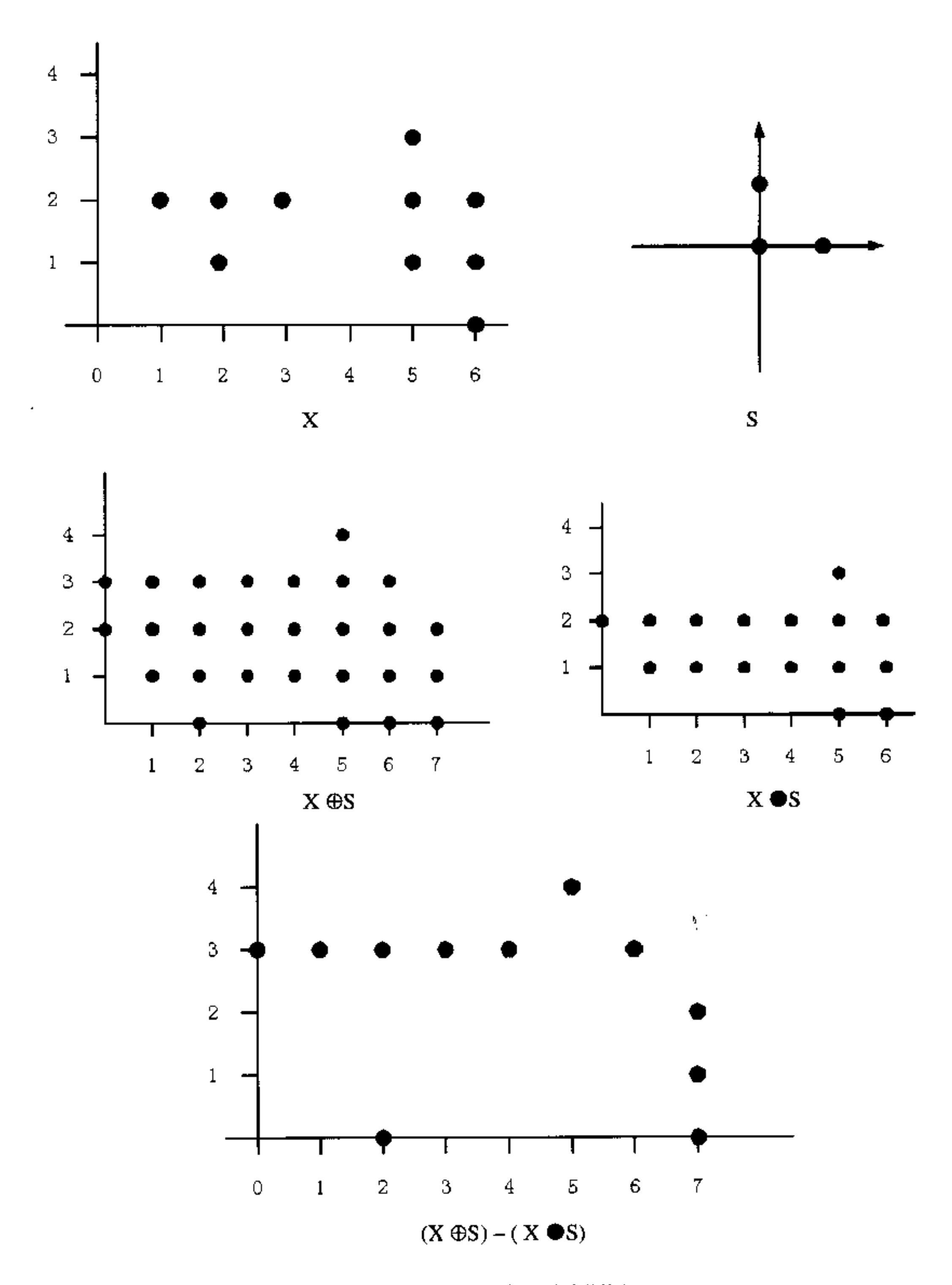


图 7－30 寻找图像在某些方向上的轮廓

7.4.2 开、闭运算的代数性质

由于开、闭运算是在腐蚀和膨胀运算的基础上定义的，根据腐蚀和膨胀运算的代数性质，我们不难得到下面的性质。

(1) 对偶性

$(X^c \circ S)^c = X \bullet S$

$(X^c \bullet S)^c = X \circ S$

(2) 扩展（收缩性）

$X \circ S \subseteq X \subseteq X \bullet S$

即开运算恒使原图像缩小，而闭运算恒使原图像扩大。

(3) 单调性

如果 $X \subseteq X'$，那么

$X \circ S \subseteq X' \circ S$

$X \bullet S \subseteq X' \bullet S$

如果 $B \subseteq C$ 且 $C \circ B = C$，那么

$X \circ B \supseteq X \circ C$

$X \circ B \supseteq X \circ C$

✎ **根据这一性质可以知道，结构元素的扩大只有在保证扩大后的结构元素对原结构元素开运算不变的条件下方能保持单调性。**

(4) 平移不变性

$X[h] \circ S = (X \circ S)[h]$

$X[h] \bullet S = (X \bullet S)[h]$

$X \circ S[h] = X \circ S$

$X \bullet S[h] = X \bullet S$

(5) 等幂性

$(X \circ S) \circ S = X \circ S$

$(X \bullet S) \bullet S = X \bullet S$

开、闭运算的等幂性是一个有趣的性质，它意味着一次滤波就能把所有特定于结构元的噪声滤除干净，作重复的运算不会再有效果。这是一个与经典方法（例如中值滤波、线性卷积）不同的性质。

7.4.3 Visual C++编程实现

开运算

开运算由函数 OpenDIB()实现。开运算相当于对图像先进行腐蚀运算再进行膨胀运算。所以在 OpenDIB()函数中，先对原图像进行腐蚀，把缓存图像中得到的结果拷贝回原图像，再对原图像进行膨胀。

```
/*************************************************************************
 *
 * 函数名称:
 *   OpenDIB()
 *
 * 参数:
```

```
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *   int   nMode     - 开运算方式，0表示水平方向，1表示垂直方向，2表示自定义结构元素。
 *   int   structure[3][3]
                          - 自定义的3×3结构元素。
 *
 * 返回值:
 *   BOOL                 - 开运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行开运算。结构元素为水平方向或垂直方向的三个点，中间点位于原点；
 * 或者由用户自己定义3×3的结构元素。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI OpenDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, int nMode , int structure[3][3])
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //循环变量
    long i;
    long j;
    int   n;
    int   m;

    //像素值
    unsigned char pixel;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);
```

```
// 初始化新分配的内存，设定初始值为255
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lWidth * lHeight);

if(nMode == 0)
{
    //使用水平方向的结构元素进行腐蚀
    for(j = 0; j <lHeight; j++)
    {
        for(i = 1;i <lWidth-1; i++)
        {
            //由于使用1×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素

            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
            if(pixel != 255 && *lpSrc != 0)
                return FALSE;

            //目标图像中的当前点先赋成黑色
            *lpDst = (unsigned char)0;

            //如果原图像中当前点自身或者左右有一个点不是黑色，
            //则将目标图像中的当前点赋成白色
            for (n = 0;n < 3;n++ )
            {
                pixel = *(lpSrc+n-1);
                if (pixel == 255 )
                {
                    *lpDst = (unsigned char)255;
                    break;
                }
            }

        }
    }
}
else if(nMode == 1)
{
    //使用垂直方向的结构元素进行腐蚀
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
```

```
                //由于使用1×3的结构元素，为防止越界，所以不处理最上边和最下边的两列像素

                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lWidth * j + i;

                // 指向目标图像倒数第j行，第i个像素的指针
                lpDst = (char *)lpNewDIBBits + lWidth * j + i;

                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                //目标图像中含有0和255外的其他灰度值
                if(pixel != 255 && *lpSrc != 0)
                    return FALSE;

                //目标图像中的当前点先赋成黑色
                *lpDst = (unsigned char)0;

                //如果原图像中当前点自身或者上下有一个点不是黑色，
                //则将目标图像中的当前点赋成白色
                for (n = 0;n < 3;n++ )
                {
                    pixel = *(lpSrc+(n-1)*lWidth);
                    if (pixel == 255 )
                    {
                        *lpDst = (unsigned char)255;
                        break;
                    }
                }

            }
        }
}
else
{
    //使用自定义的结构元素进行腐蚀
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            //由于使用3×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素
            //和最上边和最下边的两列像素
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;
```

```
                    //目标图像中含有0和255外的其他灰度值
                    if(pixel != 255 && *lpSrc != 0)
                        return FALSE;

                    //目标图像中的当前点先赋成黑色
                    *lpDst = (unsigned char)0;

                    //如果原图像中对应结构元素中为黑色的那些点中有一个不是黑色,
                    //则将目标图像中的当前点赋成白色
                    //注意在DIB图像中内容是上下倒置的
                    for (m = 0;m < 3;m++ )
                    {
                        for (n = 0;n < 3;n++)
                        {
                            if( structure[m][n] == -1)
                                continue;
                            pixel = *(lpSrc + ((2-m)-1)*lWidth + (n-1));
                            if (pixel == 255 )
                            {
                                *lpDst = (unsigned char)255;
                                break;
                            }
                        }
                    }

                }
            }

        }
        // 复制腐蚀后的图像
        memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

        // 重新初始化新分配的内存, 设定初始值为255
        //lpDst = (char *)lpNewDIBBits;
        //memset(lpDst, (BYTE)255, lWidth * lHeight);

//再进行膨胀运算
        if(nMode == 0)
        {
            //使用水平方向的结构元素进行膨胀
            for(j = 0; j <lHeight; j++)
            {
                for(i = 1;i <lWidth-1; i++)
                {
                    //由于使用1×3的结构元素, 为防止越界, 所以不处理最左边和最右边的两列像素

                    // 指向原图像倒数第j行, 第i个像素的指针
                    lpSrc = (char *)lpDIBBits + lWidth * j + i;

                    // 指向目标图像倒数第j行, 第i个像素的指针
                    lpDst = (char *)lpNewDIBBits + lWidth * j + i;
```

```
                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                //目标图像中含有0和255外的其他灰度值
                if(pixel != 255 && *lpSrc != 0)
                    return FALSE;

                //目标图像中的当前点先赋成白色
                *lpDst = (unsigned char)255;

                //原图像中当前点自身或者左右只要有一个点是黑色，
                //则将目标图像中的当前点赋成黑色
                for (n = 0;n < 3;n++ )
                {
                    pixel = *(lpSrc+n-1);
                    if (pixel == 0 )
                    {
                        *lpDst = (unsigned char)0;
                        break;
                    }
                }

            }
        }

}
else if(nMode == 1)
{
    //使用垂直方向的结构元素进行膨胀
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            //由于使用1×3的结构元素，为防止越界，所以不处理最上边和最下边的两列像素

            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
            if(pixel != 255 && *lpSrc != 0)
                return FALSE;

            //目标图像中的当前点先赋成白色
            *lpDst = (unsigned char)255;
```

```
                //原图像中当前点自身或者上下只要有一个点是黑色，
                //则将目标图像中的当前点赋成黑色
                for (n = 0;n < 3;n++ )
                {
                    pixel = *(lpSrc+(n-1)*lWidth);
                    if (pixel == 0 )
                    {
                        *lpDst = (unsigned char)0;
                        break;
                    }
                }

            }
        }

}
else
{
    //使用自定义的结构元素进行膨胀
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 0;i <lWidth; i++)
        {
            //由于使用3×3的结构元素，为防止越界，所以不处理最左边和最右边的两列像素
            //和最上边和最下边的两列像素
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
            if(pixel != 255 && *lpSrc != 0)
                return FALSE;

            //目标图像中的当前点先赋成白色
            *lpDst = (unsigned char)255;

            //原图像中对应结构元素中为黑色的那些点中只要有一个是黑色，
            //则将目标图像中的当前点赋成黑色
            //注意在DIB图像中内容是上下倒置的
            for (m = 0;m < 3;m++ )
            {
                for (n = 0;n < 3;n++)
                {
                    if( structure[m][n] == -1)
                        continue;
```

```
                        pixel = *(lpSrc + ((2-m)-1)*lWidth + (n-1));
                        if (pixel == 0 )
                        {
                            *lpDst = (unsigned char)0;
                            break;
                        }
                    }
                }

            }
        }

    }
    // 复制膨胀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);


    return TRUE;
}
```

在 ch1_1view 中加入下面的事件处理代码：

```
void CCh1_1View::OnMorphOpen()
{
    //开运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的开运算，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的开运算！ ", "系统提示" , MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
```

```
		return;
	}

	int nMode;

	// 创建对话框
	cDlgMorphOpen dlgPara;

	// 初始化变量值
	dlgPara.m_nMode = 0;

	// 显示对话框，提示用户设定开运算方向
	if (dlgPara.DoModal() != IDOK)
	{
		// 返回
		return;
	}

	// 获取用户设定的开运算方向
	nMode = dlgPara.m_nMode;

	int structure[3][3];
	if (nMode == 2)
	{
		structure[0][0]=dlgPara.m_nStructure1;
		structure[0][1]=dlgPara.m_nStructure2;
		structure[0][2]=dlgPara.m_nStructure3;
		structure[1][0]=dlgPara.m_nStructure4;
		structure[1][1]=dlgPara.m_nStructure5;
		structure[1][2]=dlgPara.m_nStructure6;
		structure[2][0]=dlgPara.m_nStructure7;
		structure[2][1]=dlgPara.m_nStructure8;
		structure[2][2]=dlgPara.m_nStructure9;
	}

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用ErosionDIB()函数开运算DIB
	if (OpenDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB), nMode ,
structure))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);
```

```
			// 更新视图
			pDoc->UpdateAllViews(NULL);
		}
		else
		{
			// 提示用户
			MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示",
MB_ICONINFORMATION | MB_OK);
		}

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 恢复光标
		EndWaitCursor();

}
```

闭运算

闭运算由函数 CloseDIB()实现。闭运算相当于对图像先膨胀再腐蚀，所以可以在OpenDIB()函数中直接调用 ErosionDIB()和 DilationDIB()函数进行腐蚀和膨胀运算。

```
/*************************************************************************
 *
 * 函数名称:
 *   CloseDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *   int   nMode          - 闭运算方式，0表示水平方向，1表示垂直方向，2表示自定义结构元素。
 *        int    structure[3][3]
 *                        - 自定义的3×3结构元素。
 *
 * 返回值:
 *   BOOL                 - 闭运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行开运算。结构元素为水平方向或垂直方向的三个点，中间点位于原点；
 * 或者由用户自己定义3×3的结构元素。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/


BOOL WINAPI CloseDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight, int nMode , int structure[3][3])
{
		if (DilationDIB(lpDIBBits, lWidth, lHeight, nMode , structure))
		{
```

```
				if (ErosionDIB(lpDIBBits, lWidth, lHeight, nMode , structure))
				{
					// 返回
					return TRUE;

				}
			}
			return FALSE;

		// 返回
		return TRUE;
}
```

ch1_1view 中的事件处理代码为：

```
void CCh1_1View::OnMorphClose()
{
		//闭运算

		// 获取文档
		CCh1_1Doc* pDoc = GetDocument();

		// 指向DIB的指针
		LPSTR	lpDIB;

		// 指向DIB像素指针
		LPSTR		lpDIBBits;

		// 锁定DIB
		lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

		// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的闭运算，其他的可以类推）
		if (::DIBNumColors(lpDIB) != 256)
		{
			// 提示用户
			MessageBox("目前只支持256色位图的闭运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

			// 解除锁定
			::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

			// 返回
			return;
		}

		int nMode;

		// 创建对话框
		cDlgMorphClose dlgPara;

		// 初始化变量值
		dlgPara.m_nMode = 0;
```

```
	// 显示对话框，提示用户设定闭运算方向
	if (dlgPara.DoModal() != IDOK)
	{
		// 返回
		return;
	}

	// 获取用户设定的闭运算方向
	nMode = dlgPara.m_nMode;

	int structure[3][3];
	if (nMode == 2)
	{
		structure[0][0]=dlgPara.m_nStructure1;
		structure[0][1]=dlgPara.m_nStructure2;
		structure[0][2]=dlgPara.m_nStructure3;
		structure[1][0]=dlgPara.m_nStructure4;
		structure[1][1]=dlgPara.m_nStructure5;
		structure[1][2]=dlgPara.m_nStructure6;
		structure[2][0]=dlgPara.m_nStructure7;
		structure[2][1]=dlgPara.m_nStructure8;
		structure[2][2]=dlgPara.m_nStructure9;
	}

	// 删除对话框
	delete dlgPara;

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用CloseDIB()函数对DIB进行闭运算
	if (CloseDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB), nMode ,
structure))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);

		// 更新视图
		pDoc->UpdateAllViews(NULL);
	}
	else
	{
		// 提示用户
		MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示" ,
MB_ICONINFORMATION | MB_OK);
	}
```

```
    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

图 7－31 和图 7－32 给出了其中的开运算运行结果。

This example shows open and close operation.

图 7－31 原图像

This example shows open and close operation.

图 7－32 用九点结构元素进行开运算的结果

7.5 数学形态学的其他运算

7.5.1 击中/击不中（Hit/Miss）变换

一般来说，一个物体的结构可以由物体内部各种成分之间的关系来确定。为了研究物体（在这里指图像）的结构，可以逐个地利用各种成分（例如各种结构元素）对其进行检验，判定哪些成分包括在图像内，哪些在图像外，从而最终确定图像的结构。击中/击不中变换就是在这个意义上提出的。

设 X 是被研究的图像，S 是结构元素，而且 S 由两个不相交的部分 S_1 和 S_2。组成，即 $S = S_1 \cup S_2$

$S_1 \cap S_2 = \Phi$

于是，X 被 S “击中” 的结果定义为

$X \otimes S = \{p \mid (S_1)_p \subseteq X \text{ 且 } (S_2)_p \subseteq X^c\}$

就是说，X 被 S 击中的结果仍是一个图像，其中每点 p 必须同时满足两个条件：S_1 被 P 平移后包含在 X 内，而且 S_2 被 P 平移后不在 X 内。图 7－33 画出了一个 X 被 S 击中的例子。击中运算还有另外一种表达形式：

$$X \otimes S = (X \ominus S_1) \cap (X^c \oplus S_2)$$
$$= (X \ominus S_1) \cap (X \oplus S_2^v)^c$$
$$= (X \ominus S_1) - (X \oplus S_2^v)$$

上式表明，X 被 S 击中的结果相当于 X 被 $S1$ 腐蚀的结果与 X 被 $S2$ 的反射集 $S2$ 膨胀的结果之差。图 7－34 解释了这一过程。由此可见，击中运算也可以借助于腐蚀、膨胀两个基本运算来实现。

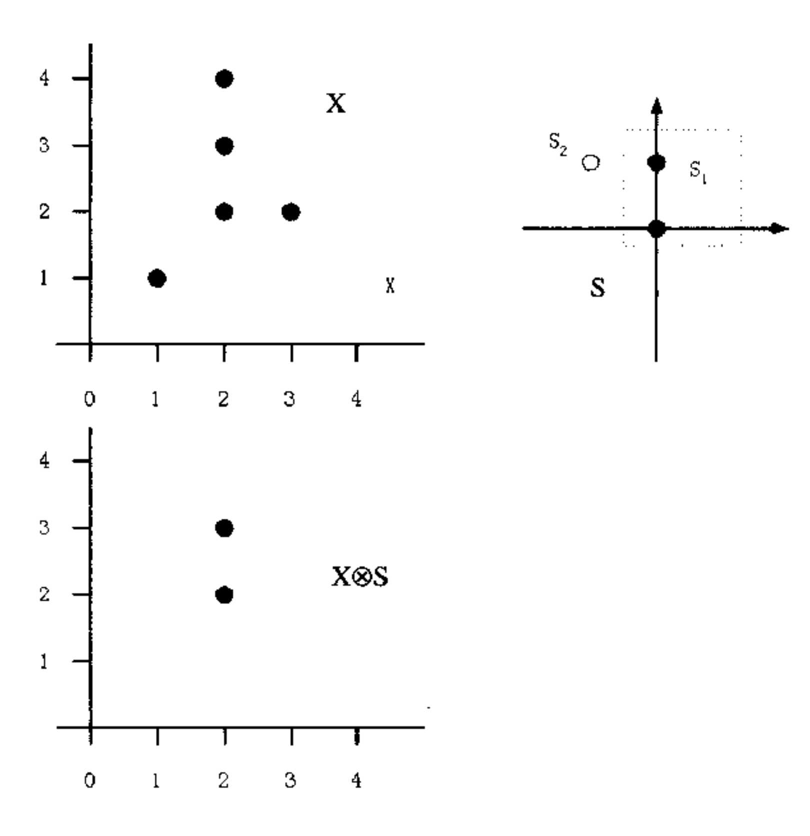


图 7－33　X 被 S 击中示意图

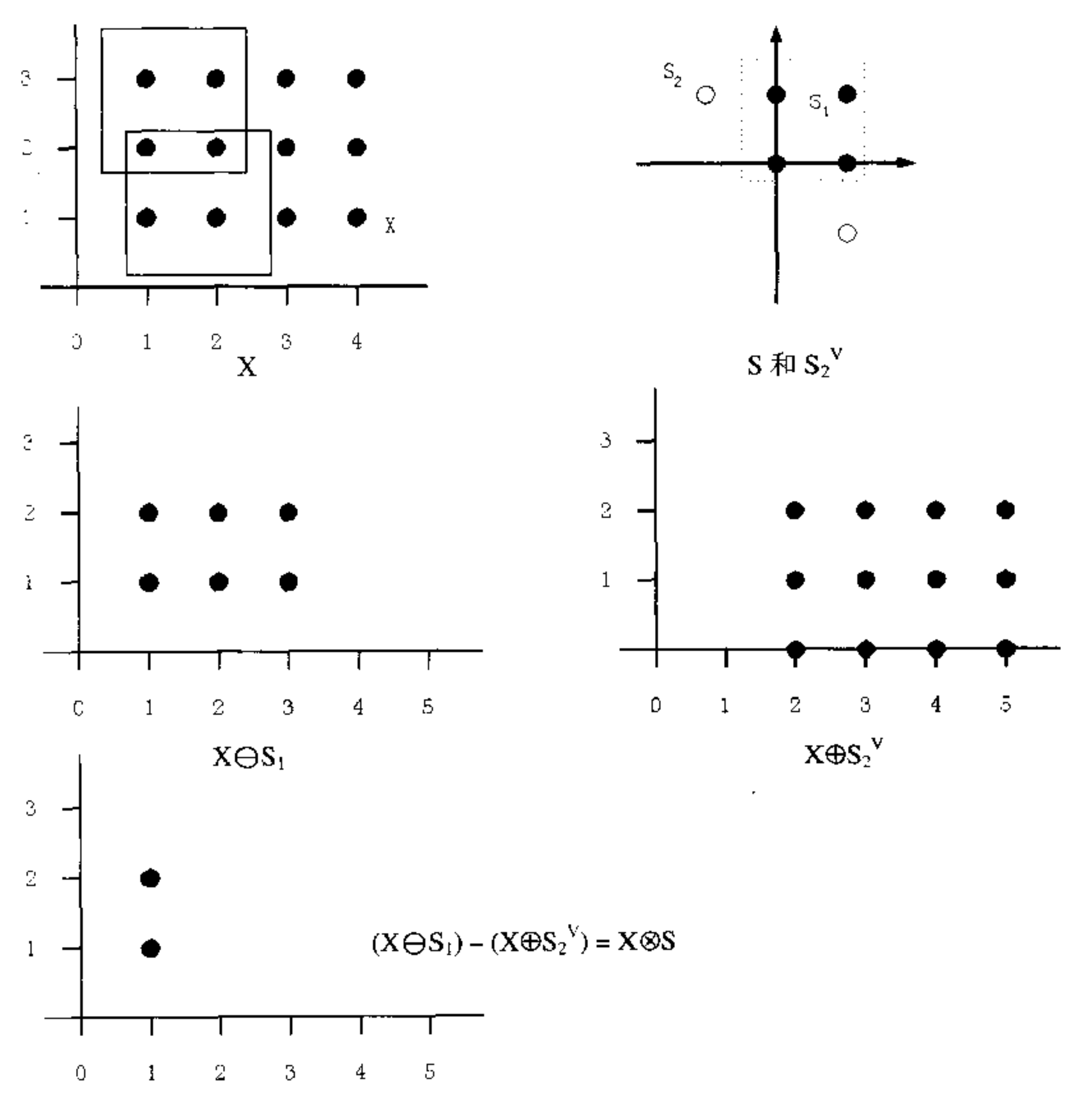


图 7－34 对击中的另一种解释

现在我们进一步来讨论击中运算的含意。在图 7－33 中，如果 S 中不包含 S_2，那么 $X\otimes S$ 与 $X\ominus S_1$ 相同，共包括 6 个点。这表明 X 被 S_1 腐蚀后还剩 6 个点，就是说，X 中共包含六个形如 S_1 的结构元素，它们的原点位置构成 $X\ominus S_1$。

将 S_2 加入 S 后，相当于对 $X\otimes S$ 增加了一个条件：不仅从 X 中找出那些形如 S_1 的部分，而且要去掉那些在左边有一个邻点的部分。这样一来，在 X 中只剩下两部分（在方框内）满足要求的结构元素。它们的原点位置构成了最终的 $X\otimes S$。

由此可见，击中运算相当于一种条件比较严格的模板匹配，它不仅指出被匹配点所应满足的性质即模板的形状，同时也指出这些点所不应满足的性质，即对周围环境背景的要求。

击中/击不中变换可以用于保持拓扑结构的形状细化，以及形状识别和定位。设有一个模板形状（集合）A，对给定的图像 X，假定 X 中有包括 A 在内的多个不同物体。我们的目的是识别和定位其中的 A 物体。此时，取一个比 A 稍大的集合 B 作为结构元素且使得 A 不与 B

的边缘相交，令 $B_1=A$ 且 $B_2=B-A$，那么 $X\otimes(B1, B2)$ 将给出且仅给出所有 X 中与 A 的误差在设定范围内的物体的位置。图 7－35 描述了一个字符识别的示例。

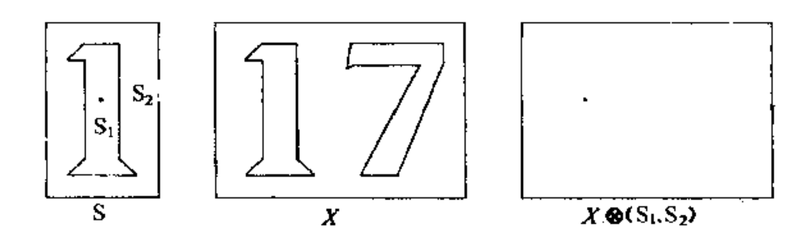


图 7－35 击中/击不中变换用于字符识别

7.5.2 细化 (Thining)

一个图像的“骨架”，是指图像中央的骨骼部分。是描述图像几何及拓扑性质的重要特征之一。求一图像骨架的过程通常称为对图像“细化”的过程。在文字识别、地质构造识别、工业零件形状识别或图像理解中，先对被处理的图像进行细化有助于突出形状特点和减少冗余的信息量。

用一个形象的比喻来说明骨架的含意。在图中，我们把图像 X 看成一块地，假定在同一时刻 $t=0$ 在地上各条边界上的每一点同时举火，则图 X 的边界上将立即出现两堵“火墙”，并向 X 的内部蔓延。再设火墙蔓延速度为常数，则在 m_0 点燃起的火经过时间 $t=|m-m_0|$ 后将蔓延到 m 点。两堵火墙相遇的地点，例如图中的 P 点，和这些点所连成的线，即图中的 $Me(X)$，便构成了图像 X 的骨架。

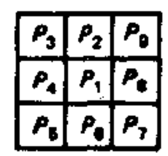
通过以上的形象说明还可以看到，在细化一幅图像 X 的过程中应满足两个条件：第一，在细化的过程中，X 应该有规律地缩小；第二，在 X 逐步缩小的过程中，应当使 X 的连通性质保持不变。

下面我们来看一个具体的细化算法。

一幅图像中的一个 3×3 区域，对各点标记名称 $P1$，$P2$，…$P8$，其中 $P1$ 位于中心。如果 $P1=1$（即黑点），下面四个条件如果同时满足，则删除 $P1$（$P1=0$）。

- $2 \leqslant NZ(P1) \leqslant 6$;
- $Z0(P1)=1$;
- $P2*P4*P8=0$ 或者 $Z0(P1) \neq 1$;
- $P2*P4*P6=0$ 或者 $Z0(P4) \neq 1$;

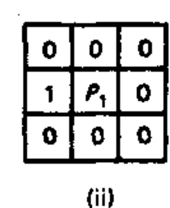
对图像中的每一个点重复这一步骤，直到所有的点都不可删除为止。图 7－36 给出了这一算法的应用示例。



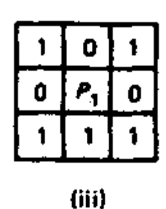
(a)标记点 P1 和邻点

1	1	0
1	P1	1
0	0	0

(i)



(ii)



(iii)

(b) 几种 P1 不可删除的情况(p1=1)

(i) 删除 P1 会分割区域;

(ii) 删除 P1 会缩短边缘;

(iii) 2≤NZ(P1)≤6，但 P1 不可删除。



(i)



(ii)

(c) 细化的结果示例。

(i) 原图像

(ii) 细化图像

图 7—36 一个细化的算法

7.5.3 Visual C++编程实现

下面给出 7.5.2 节中所述的细化算法程序。细化过程由 ThinDIB 函数实现。

```
/*************************************************************************
 *
 * 函数名称:
 *   ThinDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *
 * 返回值:
```

```
 *   BOOL                  - 闭运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行细化运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI ThiningDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //脏标记
    BOOL bModified;

    //循环变量
    long i;
    long j;
    int   n;
    int   m;

    //四个条件
    BOOL bCondition1;
    BOOL bCondition2;
    BOOL bCondition3;
    BOOL bCondition4;

    //计数器
    unsigned char nCount;

    //像素值
    unsigned char pixel;

    //5×5相邻区域像素值
    unsigned char neighbour[5][5];

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
```

```
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);

    bModified=TRUE;

    while(bModified)
    {

        bModified = FALSE;
        // 初始化新分配的内存，设定初始值为255
        lpDst = (char *)lpNewDIBBits;
        memset(lpDst, (BYTE)255, lWidth * lHeight);

        for(j = 2; j <lHeight-2; j++)
        {
            for(i = 2;i <lWidth-2; i++)
            {

                bCondition1 = FALSE;
                bCondition2 = FALSE;
                bCondition3 = FALSE;
                bCondition4 = FALSE;

                //由于使用5×5的结构元素，为防止越界，所以不处理外围的几行和几列像素

                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lWidth * j + i;

                // 指向目标图像倒数第j行，第i个像素的指针
                lpDst = (char *)lpNewDIBBits + lWidth * j + i;

                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                //目标图像中含有0和255外的其他灰度值
                if(pixel != 255 && *lpSrc != 0)
                    return FALSE;

                //如果原图像中当前点为白色，则跳过
                else if(pixel == 255)
                    continue;

                //获得当前点相邻的5×5区域内像素值，0代表白色，1代表黑色
                for (m = 0;m < 5;m++ )
```

```
{
    for (n = 0;n < 5;n++)
    {
        neighbour[m][n] =(255 - (unsigned char)*(lpSrc + ((4 - m) - 2)*lWidth + n - 2 ))
/ 255;
    }
}
//                  neighbour[][]
//逐个判断条件。
//判断2<=NZ(P1)<=6
nCount =    neighbour[1][1] + neighbour[1][2] + neighbour[1][3] \
        + neighbour[2][1] + neighbour[2][3] + \
        + neighbour[3][1] + neighbour[3][2] + neighbour[3][3];
if ( nCount >= 2 && nCount <=6)
    bCondition1 = TRUE;

//判断Z0(P1)=1
nCount = 0;
if (neighbour[1][2] == 0 && neighbour[1][1] == 1)
    nCount++;
if (neighbour[1][1] == 0 && neighbour[2][1] == 1)
    nCount++;
if (neighbour[2][1] == 0 && neighbour[3][1] == 1)
    nCount++;
if (neighbour[3][1] == 0 && neighbour[3][2] == 1)
    nCount++;
if (neighbour[3][2] == 0 && neighbour[3][3] == 1)
    nCount++;
if (neighbour[3][3] == 0 && neighbour[2][3] == 1)
    nCount++;
if (neighbour[2][3] == 0 && neighbour[1][3] == 1)
    nCount++;
if (neighbour[1][3] == 0 && neighbour[1][2] == 1)
    nCount++;
if (nCount == 1)
    bCondition2 = TRUE;

//判断P2*P4*P8=0 or Z0(p2)!=1
if (neighbour[1][2]*neighbour[2][1]*neighbour[2][3] == 0)
    bCondition3 = TRUE;
else
{
    nCount = 0;
    if (neighbour[0][2] == 0 && neighbour[0][1] == 1)
        nCount++;
    if (neighbour[0][1] == 0 && neighbour[1][1] == 1)
        nCount++;
    if (neighbour[1][1] == 0 && neighbour[2][1] == 1)
        nCount++;
    if (neighbour[2][1] == 0 && neighbour[2][2] == 1)
        nCount++;
```

```
                if (neighbour[2][2] == 0 && neighbour[2][3] == 1)
                    nCount++;
                if (neighbour[2][3] == 0 && neighbour[1][3] == 1)
                    nCount++;
                if (neighbour[1][3] == 0 && neighbour[0][3] == 1)
                    nCount++;
                if (neighbour[0][3] == 0 && neighbour[0][2] == 1)
                    nCount++;
                if (nCount != 1)
                    bCondition3 = TRUE;
            }

            //判断P2*P4*P6=0 or Z0(p4)!=1
            if (neighbour[1][2]*neighbour[2][1]*neighbour[3][2] == 0)
                bCondition4 = TRUE;
            else
            {
                nCount = 0;
                if (neighbour[1][1] == 0 && neighbour[1][0] == 1)
                    nCount++;
                if (neighbour[1][0] == 0 && neighbour[2][0] == 1)
                    nCount++;
                if (neighbour[2][0] == 0 && neighbour[3][0] == 1)
                    nCount++;
                if (neighbour[3][0] == 0 && neighbour[3][1] == 1)
                    nCount++;
                if (neighbour[3][1] == 0 && neighbour[3][2] == 1)
                    nCount++;
                if (neighbour[3][2] == 0 && neighbour[2][2] == 1)
                    nCount++;
                if (neighbour[2][2] == 0 && neighbour[1][2] == 1)
                    nCount++;
                if (neighbour[1][2] == 0 && neighbour[1][1] == 1)
                    nCount++;
                if (nCount != 1)
                    bCondition4 = TRUE;
            }
            if(bCondition1 && bCondition2 && bCondition3 && bCondition4)
            {
                *lpDst = (unsigned char)255;
                bModified = TRUE;
            }
            else
            {
                *lpDst = (unsigned char)0;
            }
        }
    }
    // 复制腐蚀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);
```

```
    }
    // 复制腐蚀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

在细化运算菜单的事件处理函数中加入下面的代码：

```
void CCh1_1View::OnMorphThining()
{
    //闭运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR      lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的闭运算，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的细化运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }


    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用ThiningDIB()函数对DIB进行闭运算
```

```
        if (ThiningDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
        {

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示",
MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();
    }
```

图 7－37 细化的结果如图 7－38 所示：

This is an example
of thining.

图 7－37 原图像

This is an example
of thining

图 7－38 细化后的图像

第八章　图像边缘检测与提取及轮廓跟踪

8.1　边缘检测

8.1.1　基本概念

利用计算机进行图像处理有两个目的：一是产生更适合人观察和识别的图像；二是希望能由计算机自动识别和理解图像。无论为了哪种目的，图像处理中关键的一步就是对包含有大量各式各样景物信息的图像进行分解。分解的最终结果是图像被分解成一些具有某种特征的最小成分，称为图像的基元。相对于整幅图像来说，这种基元更容易被快速处理。

图像的特征指图像场中可用作标志的属性。它可以分为图像的统计特征和图像的视觉特征两类。图像的统计特征是指一些人为定义的特征，通过变换才能得到，如图像的直方图、矩、频谱等等;图像的视觉特征是指人的视觉可直接感受到的自然特征，如区域的亮度、纹理或轮廓等。利用这两类特征把图像分解成一系列有意义的目标或区域的过程称为图像的分割。

图像的边缘是图像的最基本特征。所谓边缘（或边沿）是指其周围像素灰度有阶跃变化或屋顶变化的那些像素的集合。边缘广泛存在于物体与背景之间、物体与物体之间、基元与基元之间。因此，它是图像分割所依赖的重要特征。在本节中，我们将介绍图像边缘的检测和提取技术。

物体的边缘是由灰度不连续性所反映的。经典的边缘提取方法是考察图像的每个像素在某个邻域内灰度的变化，利用边缘邻近一阶或二阶方向导数变化规律．用简单的方法检测边缘。这种方法称为边缘检测局部算子法。

边缘的种类可以分为两种：一种称为阶跃性边缘，它两边的像素的灰度值有着显著的不同；另一种称为屋顶状边缘，它位于灰度值从增加到减少的变化转折点。图 8－1 中分别给出了这两种边缘的示意图及相应的一阶方向导数、二阶方向导数的变化规律。对于阶跃性边缘，二阶方向导数在边缘处呈零交叉；而对于屋顶状边缘，二阶方向导数在边缘处取极值。

如果一个像素落在图像中某一个物体的边界上，那么它的邻域将成为一个灰度级的变化带。对这种变化最有用的两个特征是灰度的变化率和方向，它们分别以梯度向量的幅度和方向来表示。

边缘检测算子检查每个像素的邻域并对灰度变化率进行量化，也包括方向的确定。大多数使用基于方向导数掩模求卷积的方法。

下面介绍几种常用的边缘检测算子。

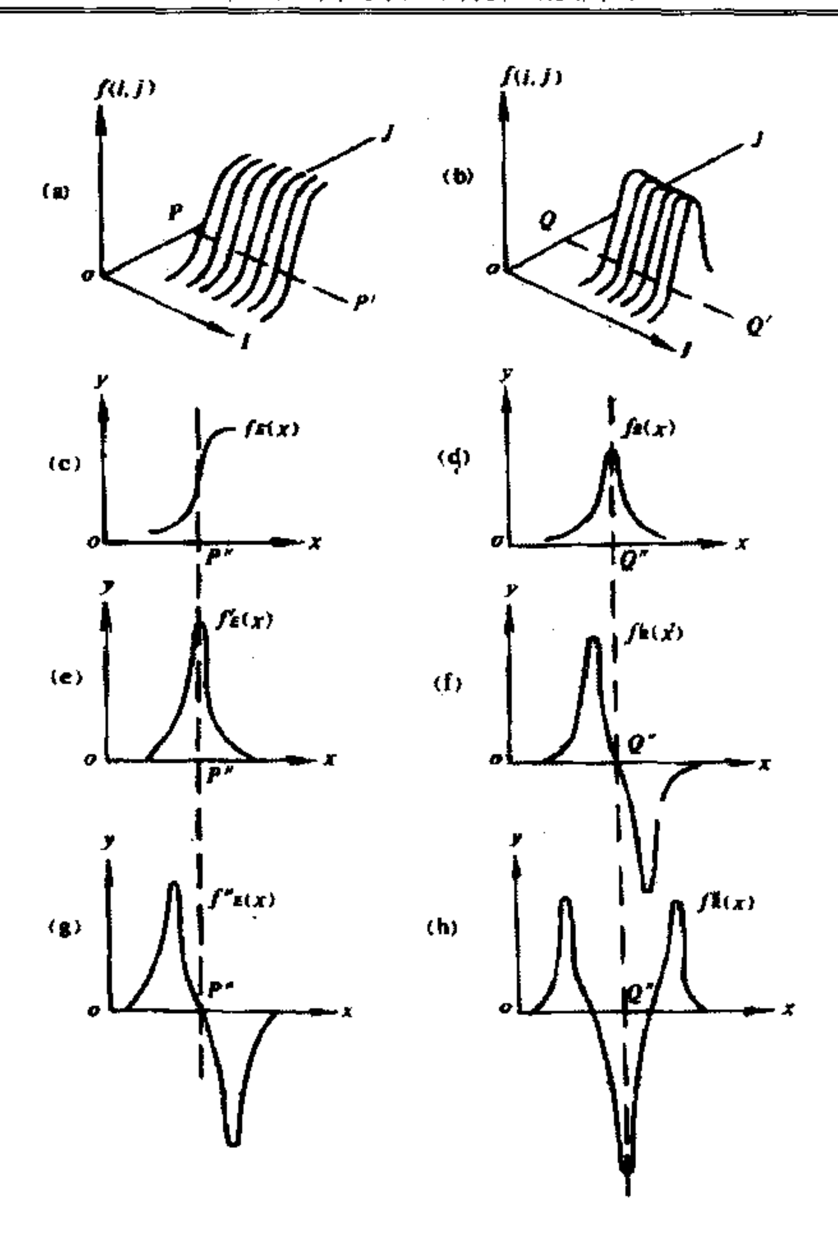


图 8－1 阶跃性边缘和屋顶状边缘处一阶及二阶导数变化规律

- Roberts 边缘检测算子

Roberts 边缘检测算子是一种利用局部差分算子寻找边缘的算子。它由下式给出：

$$g(x, y) = \{[\sqrt{f(x, y)} - \sqrt{f(x+1, y+1)}]^2 + [\sqrt{f(x, y)} - \sqrt{f(x+1, y+1)}]^2\}^{1/2}$$

其中 $f(x,y)$ 是具有整数像素坐标的输入图像，平方根运算使该处理类似于在人类视觉系统中发生的过程。

- Sobel 边缘算子

图 8－2 所示的两个卷积核形成了 sobel 边缘算子，图像中的每个点都用这两个核做卷积，

一个核对通常的垂直边缘响应最大，而另一个对水平边缘响应最大。两个卷积的最大值作为该点的输出位。运算结果是一幅边缘幅度图像。

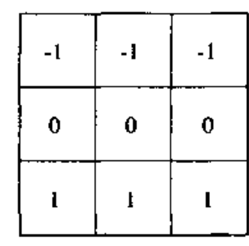
-1	-2	-1
0	0	0
1	2	1

-1	0	1
-2	0	2
-1	0	1

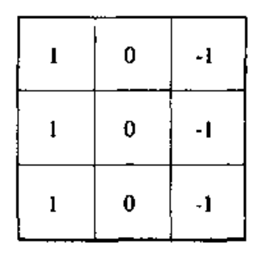
图 8－2 Sobel 边缘检测算子

- Prewitt 边缘算子

图 8－3 所示的两个卷积核形成了 Prewitt 边缘算子。和使用 Sobel 算子的方法一样，图像中的每个点都用这两个核进行卷积，取最大值作为输出。Prewitt 算子也产生一幅边缘幅度图像。



-1	-1	-1
0	0	0
1	1	1



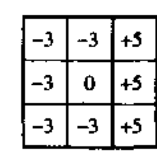
1	0	-1
1	0	-1
1	0	-1

图 8－3 Prewitt 边缘检测算子

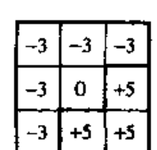
- Krisch 边缘算子

图 8－4 所示的 8 个卷积核组成了 Kirsch 边缘算子。图像中的每个点都用 8 个掩模进行卷积，每个掩模都对某个特定边缘方向作出最大响应，所有 8 个方向中的最大值作为边缘幅度图像的输出。最大响应掩模的序号构成了边缘方向的编码。

+5	+5	+5
-3	0	-3
-3	-3	-3



-3	+5	+5
-3	0	+5
-3	-3	-3



-3	-3	+5
-3	0	+5
-3	-3	+5

-3	-3	-3
-3	0	+5
-3	+5	+5

-3	-3	-3
-3	0	-3
+5	+5	+5

-3	-3	-3
+5	0	-3
+5	+5	-3

+5	-3	-3
+5	0	-3
+5	-3	-3

+5	+5	-3
+5	0	-3
-3	-3	-3

图 8－4 Kirsch 边缘算子

● 高斯一拉普拉斯算子

拉普拉斯算子是对二维函数进行运算的二阶导数算子。通常使用的拉普拉斯算子如图8－5所示。

0	-1	0
-1	4	-1
0	-1	0

-1	-1	-1
-1	8	-1
-1	-1	-1

图8－5 拉普拉斯边缘检测算子

由于拉普拉斯算子是一个二阶导数，它将在边缘处产生一个陡峭的零交叉，如图8－1所示。

由于噪声点对边沿检测有一定的影响，所以高斯拉普拉斯算子是效果较好的边沿检测器。它把高斯平滑滤波器和拉普拉斯锐化滤波器结合了起来，先平滑掉噪声，再进行边沿检测，所以效果更好。常用的高斯拉普拉斯算子是5×5的模板：

$$\begin{bmatrix} -2 & -4 & -4 & -4 & -2 \\ -4 & 0 & 8 & 0 & -4 \\ -4 & 8 & 24 & 8 & -4 \\ -4 & 0 & 8 & 0 & -4 \\ -2 & -4 & -4 & -4 & -2 \end{bmatrix}$$

它的脉冲响应和传递函数如图8－6所示。

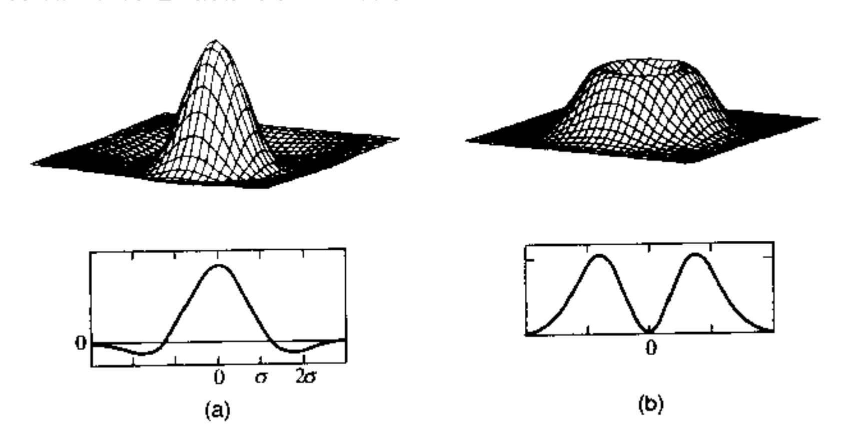


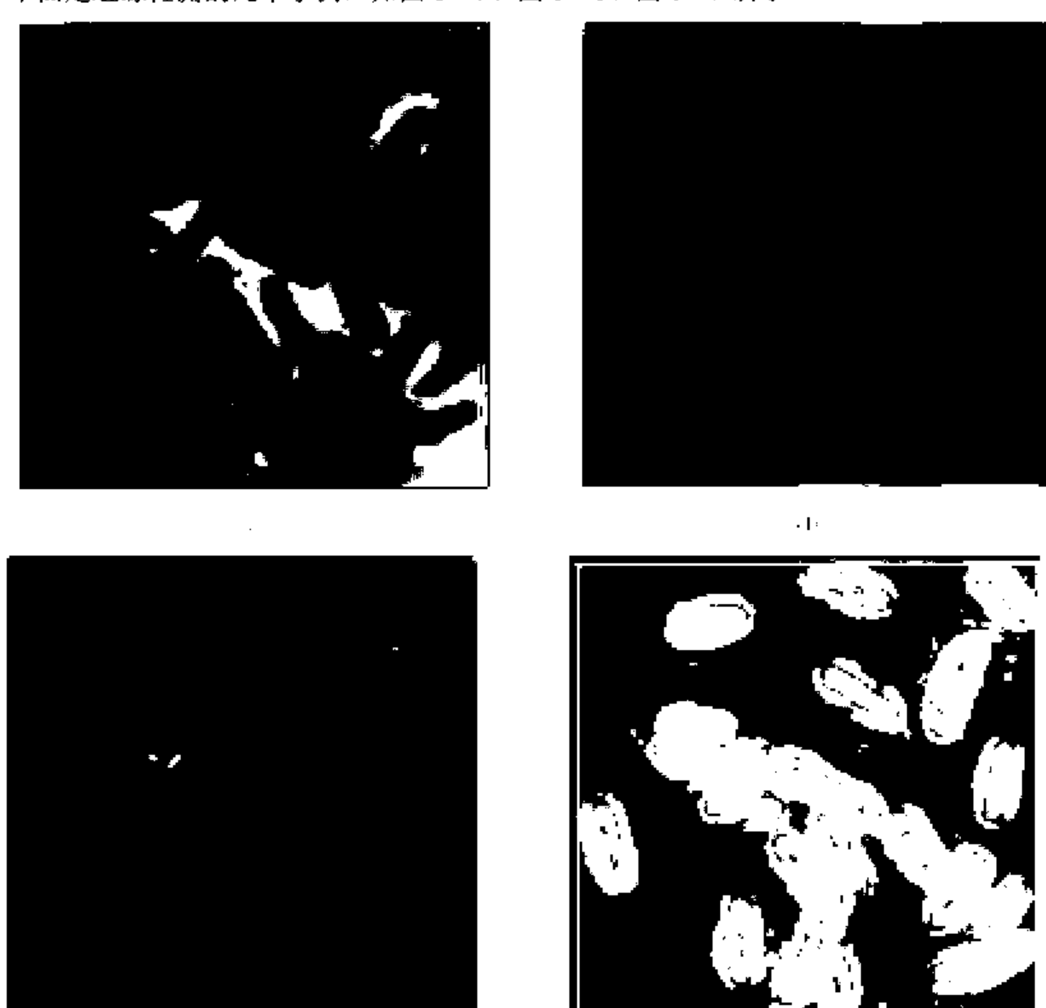
图8－6 高斯拉普拉斯算子 (a) 脉冲响应 (b) 传递函数

● 边缘检测器性能

上述边缘算子产生的边缘图像看来很相似，因此它们看起来像一个绘画者从图片中作出的线条画。Robert算子是2×2算子，对具有陡峭的低噪声图像响应最好。其他三个算子都是3×3算子，对灰度渐变和噪声较多的图像处理得较好。

- **使用两个掩模板组成边缘检测器时，通常取两个掩模检测所得结果中幅度较大的作为输出值。这使得它们对边缘的走向有些敏感，取它们的平方和的开方可以获得性能更一致的与真实的梯度值更接近的全方位响应。**

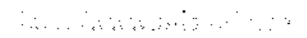
下面是边缘检测的几个示例，如图 8—7、图 8—8、图 8—9 所示。

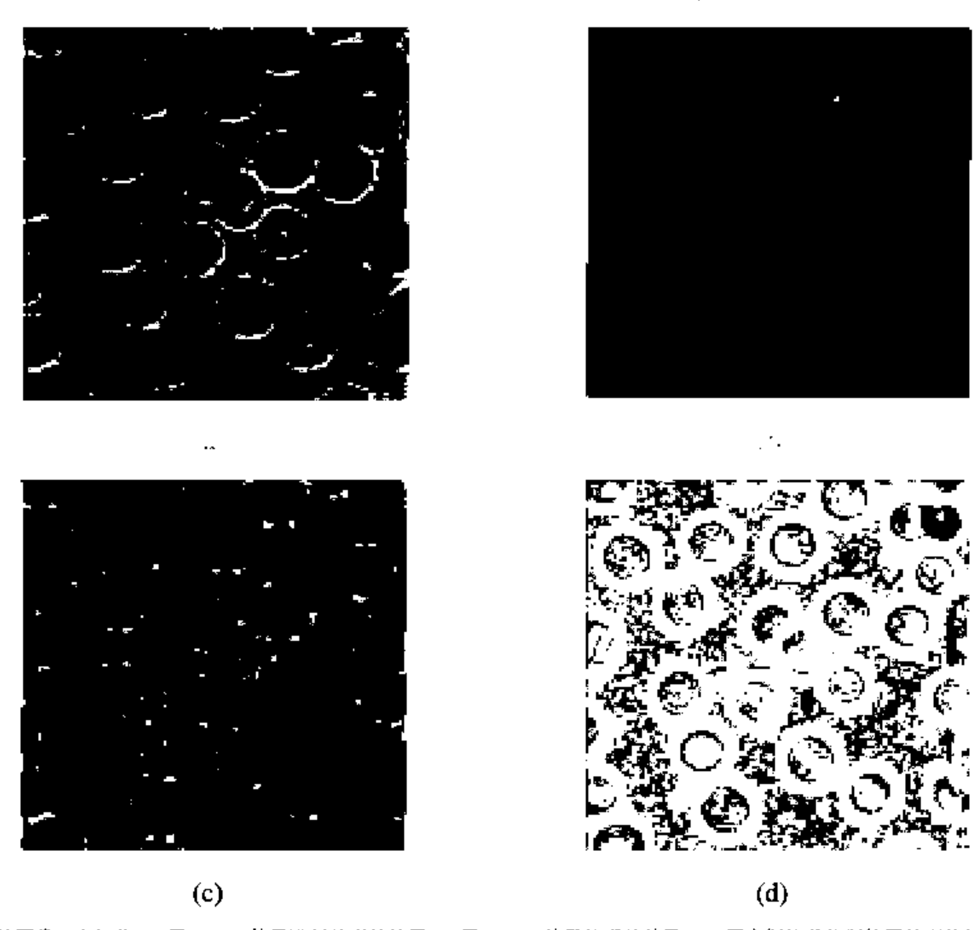


(a) 原始图像：显微镜下看到的细菌 (b)用 robert 算子进行处理的结果 (c)用 prewitt 算子处理的结果 (d) 用高斯拉普拉斯算子处理的结果

图 8—7 边缘检测示例

- **由于 prewitt 算子并不是各向同性的，所以(c)中我们看到的边缘并不是完全连通的，有一定程度的断开；而使用 robert 和高斯拉普拉斯算子就不存在这样的问题。在图 8-9 中我们将看到 sobel 算子也存在类似的问题。解决这个问题的方法是把它扩展成八个方向的 sobel 和 Prewitt 边缘算子，并且可以像使用 Kirsch 算子一样获得边缘方向图；或者使用各向同性的 Isotropic Sobel 算子。**

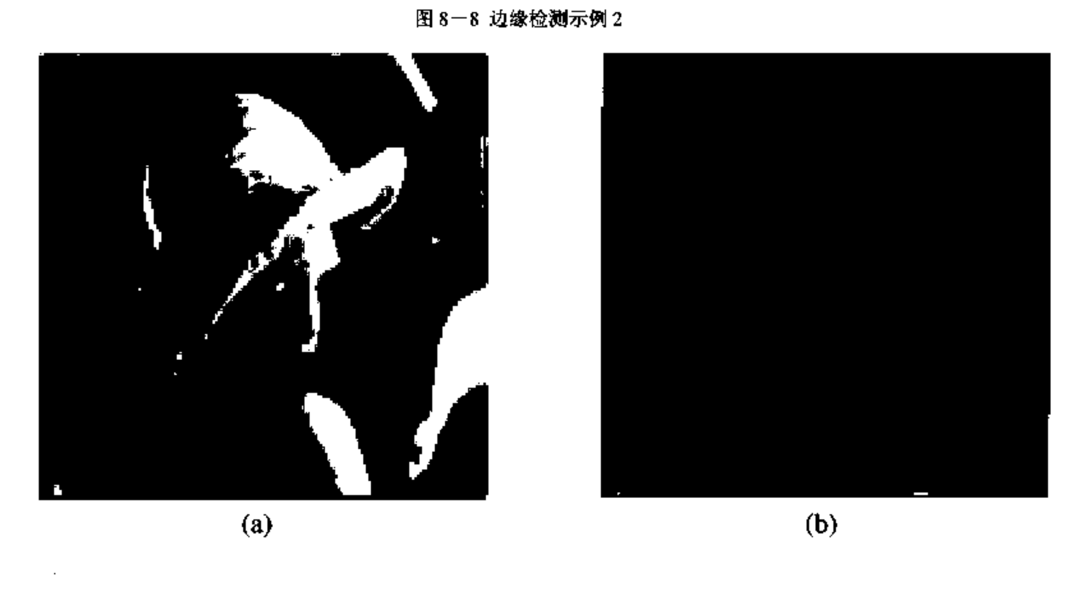




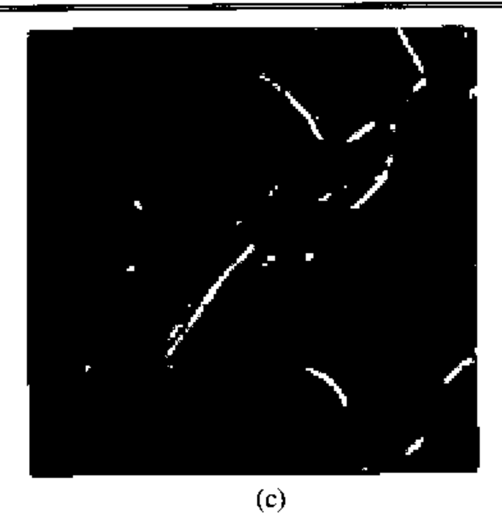
(c) (d)

(a) 原始图像：血细胞 (b)用 robert 算了进行处理的结果 (c)用 kirsch 算子处理的结果 (d) 用高斯拉普拉斯算子处理的结果

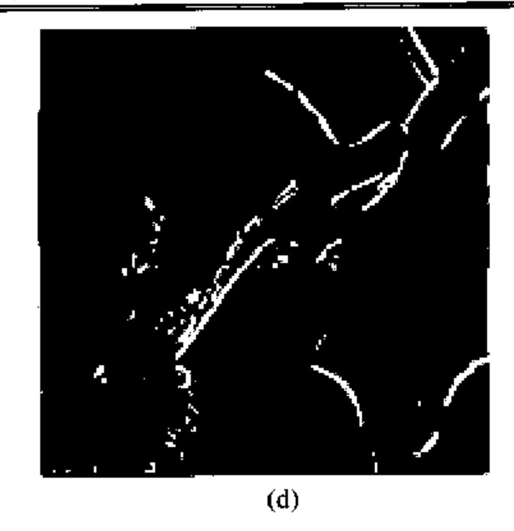
图 8－8 边缘检测示例 2



(a) (b)



(c)



(d)

(a) 原始图像 (b)用 robert 算子处理的结果 (c) prewitt 算子处理的结果 (d) 用 sobel 算子处理的结果

图 8－9 边缘检测示例 3

8.1.2 Visual C++编程实现

除了 robert 算子外其他边缘检测的算法都是将一个模板算子作用于图像，所以在编程实现的时候我们可以使用在图像增强一章中介绍过的通用的图像模板操作函数。对图像进行边缘检测操作的函数是 edgecontour.cpp 中的 RobertDIB()，SobelDIB()，PrewittDIB()，KirschDIB() 和 GaussDIB()函数。

```
// ***************************************************************************
//  文件名：edgecontour.cpp
//
//  图像边缘与轮廓运算API函数库：
//
//  RobertDIB()        - robert边缘检测运算
//  SobelDIB()         - sobel边缘检测运算
//  PrewittDIB()  - prewitt边缘检测运算
//  KirschDIB()        - kirsch边缘检测运算
//  GaussDIB()         - gauss边缘检测运算
//  HoughDIB()         - 利用Hough变换检测平行直线
//  ContourDIB()  - 轮廓提取
//  TraceDIB()          - 轮廓跟踪
//  FillDIB()      - 种子填充
//
// ***************************************************************************

#include "stdafx.h"
#include "edgecontour.h"
#include "TemplateTrans.h"
#include "DIBAPI.h"
#include <math.h>
#include <direct.h>
```

```
/*************************************************************************
 *
 * 函数名称:
 *   RobertDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度(像素数，必须是4的倍数)
 *   LONG  lHeight        - 原图像高度(像素数)
 * 返回值:
 *   BOOL                 - 边缘检测成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用Robert边缘检测算子对图像进行边缘检测运算。
 *
 * 要求目标图像为灰度图像。
 ************************************************************************/

BOOL WINAPI RobertDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //循环变量
    long i;
    long j;

    //像素值
    double result;
    unsigned char pixel[4];

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);
```

```
    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);


    //使用水平方向的结构元素进行腐蚀
    for(j = lHeight-1; j > 0; j--)
    {
        for(i = 0;i <lWidth-1; i++)
        {
            //由于使用2×2的模板，为防止越界，所以不处理最下边和最右边的两列像素

            // 指向原图像第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处2*2区域的像素值，注意要转换为unsigned char型
            pixel[0] = (unsigned char)*lpSrc;
            pixel[1] = (unsigned char)*(lpSrc + 1);
            pixel[2] = (unsigned char)*(lpSrc - lWidth);
            pixel[3] = (unsigned char)*(lpSrc - lWidth + 1);

            //计算目标图像中的当前点
/            result = sqrt(( pixel[0] - pixel[3] )*( pixel[0] - pixel[3] ) + \
                    ( pixel[1] - pixel[2] )*( pixel[1] - pixel[2] ));
            *lpDst = (unsigned char)result;

        }
    }

    // 复制腐蚀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   SobelDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
```

```
 *   LONG  lHeight       - 原图像高度（像素数）
 * 返回值:
 *   BOOL                - 边缘检测成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用Sobel边缘检测算子对图像进行边缘检测运算。
 *
 * 要求目标图像为灰度图像。
 ************************************************************************/

BOOL WINAPI SobelDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向缓存图像的指针
    LPSTR   lpDst1;
    LPSTR   lpDst2;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits1;
    HLOCAL      hNewDIBBits1;
    LPSTR   lpNewDIBBits2;
    HLOCAL      hNewDIBBits2;

    //循环变量
    long i;
    long j;

    // 模板高度
    int         iTempH;

    // 模板宽度
    int         iTempW;

    // 模板系数
    FLOAT   fTempC;

    // 模板中心元素X坐标
    int         iTempMX;

    // 模板中心元素Y坐标
    int         iTempMY;

    //模板数组
    FLOAT aTemplate[9];

    // 暂时分配内存，以保存新图像
    hNewDIBBits1 = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits1 == NULL)
    {
        // 分配内存失败
```

```
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits1 = (char * )LocalLock(hNewDIBBits1);

    // 暂时分配内存，以保存新图像
    hNewDIBBits2 = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits2 == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits2 = (char * )LocalLock(hNewDIBBits2);

    // 拷贝原图像到缓存图像中
    lpDst1 = (char *)lpNewDIBBits1;
    memcpy(lpNewDIBBits1, lpDIBBits, lWidth * lHeight);
    lpDst2 = (char *)lpNewDIBBits2;
    memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

    // 设置Sobel模板参数
    iTempW = 3;
    iTempH = 3;
    fTempC = 1.0;
    iTempMX = 1;
    iTempMY = 1;
    aTemplate[0] = -1.0;
    aTemplate[1] = -2.0;
    aTemplate[2] = -1.0;
    aTemplate[3] = 0.0;
    aTemplate[4] = 0.0;
    aTemplate[5] = 0.0;
    aTemplate[6] = 1.0;
    aTemplate[7] = 2.0;
    aTemplate[8] = 1.0;

    // 调用Template()函数
    if (!Template(lpNewDIBBits1, lWidth, lHeight,
        iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
    {
        return FALSE;
    }

    // 设置Sobel模板参数
    aTemplate[0] = -1.0;
    aTemplate[1] = 0.0;
    aTemplate[2] = 1.0;
```

```
	aTemplate[3] = -2.0;
	aTemplate[4] = 0.0;
	aTemplate[5] = 2.0;
	aTemplate[6] = -1.0;
	aTemplate[7] = 0.0;
	aTemplate[8] = 1.0;

	// 调用Template()函数
	if (!Template(lpNewDIBBits2, lWidth, lHeight,
		iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
	{
		return FALSE;
	}

	//求两幅缓存图像的最大值
	for(j = 0; j <lHeight; j++)
	{
		for(i = 0;i <lWidth-1; i++)
		{

			// 指向缓存图像1倒数第j行，第i个像素的指针
			lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

			// 指向缓存图像2倒数第j行，第i个像素的指针
			lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

			if(*lpDst2 > *lpDst1)
				*lpDst1 = *lpDst2;

		}
	}

	// 复制经过模板运算后的图像到原图像
	memcpy(lpDIBBits, lpNewDIBBits1, lWidth * lHeight);

	// 释放内存
	LocalUnlock(hNewDIBBits1);
	LocalFree(hNewDIBBits1);

	LocalUnlock(hNewDIBBits2);
	LocalFree(hNewDIBBits2);
	// 返回
	return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   PrewittDIB()
 *
 * 参数:
```

```
 *   LPSTR lpDIBBits       - 指向原DIB图像指针
 *   LONG  lWidth          - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight         - 原图像高度（像素数）
 * 返回值:
 *   BOOL                  - 边缘检测成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用Prewitt边缘检测算子对图像进行边缘检测运算。
 *
 * 要求目标图像为灰度图像。
 ************************************************************************/

BOOL WINAPI PrewittDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

	// 指向缓存图像的指针
	LPSTR	lpDst1;
	LPSTR	lpDst2;

	// 指向缓存DIB图像的指针
	LPSTR	lpNewDIBBits1;
	HLOCAL	hNewDIBBits1;
	LPSTR	lpNewDIBBits2;
	HLOCAL	hNewDIBBits2;

	//循环变量
	long i;
	long j;

	// 模板高度
	int		iTempH;

	// 模板宽度
	int		iTempW;

	// 模板系数
	FLOAT	fTempC;

	// 模板中心元素X坐标
	int		iTempMX;

	// 模板中心元素Y坐标
	int		iTempMY;

	//模板数组
	FLOAT aTemplate[9];

	// 暂时分配内存，以保存新图像
	hNewDIBBits1 = LocalAlloc(LHND, lWidth * lHeight);

	if (hNewDIBBits1 == NULL)
```

```
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits1 = (char * )LocalLock(hNewDIBBits1);

// 暂时分配内存，以保存新图像
hNewDIBBits2 = LocalAlloc(LHND, lWidth * lHeight);

if (hNewDIBBits2 == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits2 = (char * )LocalLock(hNewDIBBits2);

// 拷贝原图像到缓存图像中
lpDst1 = (char *)lpNewDIBBits1;
memcpy(lpNewDIBBits1, lpDIBBits, lWidth * lHeight);
lpDst2 = (char *)lpNewDIBBits2;
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Prewitt模板参数
iTempW = 3;
iTempH = 3;
fTempC = 1.0;
iTempMX = 1;
iTempMY = 1;
aTemplate[0] = -1.0;
aTemplate[1] = -1.0;
aTemplate[2] = -1.0;
aTemplate[3] = 0.0;
aTemplate[4] = 0.0;
aTemplate[5] = 0.0;
aTemplate[6] = 1.0;
aTemplate[7] = 1.0;
aTemplate[8] = 1.0;

// 调用Template()函数
if (!Template(lpNewDIBBits1, lWidth, lHeight,
     iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

// 设置Prewitt模板参数
aTemplate[0] = 1.0;
```

```
        aTemplate[1] = 0.0;
        aTemplate[2] = -1.0;
        aTemplate[3] = 1.0;
        aTemplate[4] = 0.0;
        aTemplate[5] = -1.0;
        aTemplate[6] = 1.0;
        aTemplate[7] = 0.0;
        aTemplate[8] = -1.0;

        // 调用Template()函数
        if (!Template(lpNewDIBBits2, lWidth, lHeight,
            iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
        {
            return FALSE;
        }

        //求两幅缓存图像的最大值
        for(j = 0; j <lHeight; j++)
        {
            for(i = 0;i <lWidth-1; i++)
            {

                // 指向缓存图像1倒数第j行，第i个像素的指针
                lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

                // 指向缓存图像2倒数第j行，第i个像素的指针
                lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

                if(*lpDst2 > *lpDst1)
                    *lpDst1 = *lpDst2;

            }
        }

        // 复制经过模板运算后的图像到原图像
        memcpy(lpDIBBits, lpNewDIBBits1, lWidth * lHeight);

        // 释放内存
        LocalUnlock(hNewDIBBits1);
        LocalFree(hNewDIBBits1);

        LocalUnlock(hNewDIBBits2);
        LocalFree(hNewDIBBits2);
        // 返回
        return TRUE;
}

/*************************************************************************
 *
 * 函数名称：
 *   KirschDIB()
```

```
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth          - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight         - 原图像高度（像素数）
 * 返回值:
 *   BOOL                   - 边缘检测成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用kirsch边缘检测算子对图像进行边缘检测运算。
 *
 * 要求目标图像为灰度图像。
 ************************************************************************/

BOOL WINAPI KirschDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向缓存图像的指针
    LPSTR   lpDst1;
    LPSTR   lpDst2;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits1;
    HLOCAL      hNewDIBBits1;
    LPSTR   lpNewDIBBits2;
    HLOCAL      hNewDIBBits2;

    //循环变量
    long i;
    long j;

    // 模板高度
    int         iTempH;

    // 模板宽度
    int         iTempW;

    // 模板系数
    FLOAT   fTempC;

    // 模板中心元素X坐标
    int         iTempMX;

    // 模板中心元素Y坐标
    int         iTempMY;

    //模板数组
    FLOAT aTemplate[9];

    // 暂时分配内存，以保存新图像
    hNewDIBBits1 = LocalAlloc(LHND, lWidth * lHeight);
```

```
if (hNewDIBBits1 == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits1 = (char * )LocalLock(hNewDIBBits1);

// 暂时分配内存，以保存新图像
hNewDIBBits2 = LocalAlloc(LHND, lWidth * lHeight);

if (hNewDIBBits2 == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits2 = (char * )LocalLock(hNewDIBBits2);

// 拷贝原图像到缓存图像中
lpDst1 = (char *)lpNewDIBBits1;
memcpy(lpNewDIBBits1, lpDIBBits, lWidth * lHeight);
lpDst2 = (char *)lpNewDIBBits2;
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);


// 设置Kirsch模板1参数
iTempW = 3;
iTempH = 3;
fTempC = 1.0;
iTempMX = 1;
iTempMY = 1;
aTemplate[0] = 5.0;
aTemplate[1] = 5.0;
aTemplate[2] = 5.0;
aTemplate[3] = -3.0;
aTemplate[4] = 0.0;
aTemplate[5] = -3.0;
aTemplate[6] = -3.0;
aTemplate[7] = -3.0;
aTemplate[8] = -3.0;

// 调用Template()函数
if (!Template(lpNewDIBBits1, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}
```

```
// 设置Kirsch模板2参数
aTemplate[0] = -3.0;
aTemplate[1] = 5.0;
aTemplate[2] = 5.0;
aTemplate[3] = -3.0;
aTemplate[4] = 0.0;
aTemplate[5] = 5.0;
aTemplate[6] = -3.0;
aTemplate[7] = -3.0;
aTemplate[8] = -3.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

//求两幅缓存图像的最大值
for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth-1; i++)
    {

        // 指向缓存图像1倒数第j行，第i个像素的指针
        lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

        // 指向缓存图像2倒数第j行，第i个像素的指针
        lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

        if(*lpDst2 > *lpDst1)
            *lpDst1 = *lpDst2;

    }
}

// 拷贝原图像到缓存图像中
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Kirsch模板3参数
aTemplate[0] = -3.0;
aTemplate[1] = -3.0;
aTemplate[2] = 5.0;

aTemplate[3] = -3.0;
aTemplate[4] = 0.0;
aTemplate[5] = 5.0;

aTemplate[6] = -3.0;
aTemplate[7] = -3.0;
```

```
aTemplate[8] = 5.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

//求两幅缓存图像的最大值
for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth-1; i++)
    {

        // 指向缓存图像1倒数第j行，第i个像素的指针
        lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

        // 指向缓存图像2倒数第j行，第i个像素的指针
        lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

        if(*lpDst2 > *lpDst1)
            *lpDst1 = *lpDst2;

    }
}

// 拷贝原图像到缓存图像中
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Kirsch模板4参数
aTemplate[0] = -3.0;
aTemplate[1] = -3.0;
aTemplate[2] = -3.0;
aTemplate[3] = -3.0;
aTemplate[4] = 0.0;
aTemplate[5] = 5.0;
aTemplate[6] = -3.0;
aTemplate[7] = 5.0;
aTemplate[8] = 5.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

//求两幅缓存图像的最大值
for(j = 0; j <lHeight; j++)
{
```

```
    for(i = 0;i <lWidth-1; i++)
    {

        // 指向缓存图像1倒数第j行，第i个像素的指针
        lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

        // 指向缓存图像2倒数第j行，第i个像素的指针
        lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

        if(*lpDst2 > *lpDst1)
            *lpDst1 = *lpDst2;

    }
}

// 拷贝原图像到缓存图像中
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Kirsch模板5参数
aTemplate[0] = -3.0;
aTemplate[1] = -3.0;
aTemplate[2] = -3.0;
aTemplate[3] = -3.0;
aTemplate[4] = 0.0;
aTemplate[5] = -3.0;
aTemplate[6] = 5.0;
aTemplate[7] = 5.0;
aTemplate[8] = 5.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

// 拷贝原图像到缓存图像中
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);
//求两幅缓存图像的最大值
for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth-1; i++)
    {

        // 指向缓存图像1倒数第j行，第i个像素的指针
        lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

        // 指向缓存图像2倒数第j行，第i个像素的指针
        lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

        if(*lpDst2 > *lpDst1)
```

```
				*lpDst1 = *lpDst2;

		}
	}

	// 拷贝原图像到缓存图像中
	memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

	// 设置Kirsch模板6参数
	aTemplate[0] = -3.0;
	aTemplate[1] = -3.0;
	aTemplate[2] = -3.0;
	aTemplate[3] = 5.0;
	aTemplate[4] = 0.0;
	aTemplate[5] = -3.0;
	aTemplate[6] = 5.0;
	aTemplate[7] = 5.0;
	aTemplate[8] = -3.0;

	// 调用Template()函数
	if (!Template(lpNewDIBBits2, lWidth, lHeight,
		iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
	{
		return FALSE;
	}

	//求两幅缓存图像的最大值
	for(j = 0; j <lHeight; j++)
	{
		for(i = 0;i <lWidth-1; i++)
		{

			// 指向缓存图像1倒数第j行，第i个像素的指针
			lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

			// 指向缓存图像2倒数第j行，第i个像素的指针
			lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

			if(*lpDst2 > *lpDst1)
				*lpDst1 = *lpDst2;

		}
	}

	// 拷贝原图像到缓存图像中
	memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

	// 设置Kirsch模板7参数
	aTemplate[0] = 5.0;
	aTemplate[1] = -3.0;
	aTemplate[2] = -3.0;
```

```
aTemplate[3] = 5.0;
aTemplate[4] = 0.0;
aTemplate[5] = -3.0;
aTemplate[6] = 5.0;
aTemplate[7] = -3.0;
aTemplate[8] = -3.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}

//求两幅缓存图像的最大值
for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth-1; i++)
    {

        // 指向缓存图像1倒数第j行，第i个像素的指针
        lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

        // 指向缓存图像2倒数第j行，第i个像素的指针
        lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

        if(*lpDst2 > *lpDst1)
            *lpDst1 = *lpDst2;

    }
}

// 拷贝源图像到缓存图像中
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Kirsch模板8参数
aTemplate[0] = 5.0;
aTemplate[1] = 5.0;
aTemplate[2] = -3.0;
aTemplate[3] = 5.0;
aTemplate[4] = 0.0;
aTemplate[5] = -3.0;
aTemplate[6] = -3.0;
aTemplate[7] = -3.0;
aTemplate[8] = -3.0;

// 调用Template()函数
if (!Template(lpNewDIBBits2, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
```

```
    }

    //求两幅缓存图像的最大值
    for(j = 0; j <lHeight; j++)
    {
        for(i = 0;i <lWidth-1; i++)
        {

            // 指向缓存图像1倒数第j行，第i个像素的指针
            lpDst1 = (char *)lpNewDIBBits1 + lWidth * j + i;

            // 指向缓存图像2倒数第j行，第i个像素的指针
            lpDst2 = (char *)lpNewDIBBits2 + lWidth * j + i;

            if(*lpDst2 > *lpDst1)
                *lpDst1 = *lpDst2;

        }
    }

    // 复制经过模板运算后的图像到原图像
    memcpy(lpDIBBits, lpNewDIBBits1, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits1);
    LocalFree(hNewDIBBits1);

    LocalUnlock(hNewDIBBits2);
    LocalFree(hNewDIBBits2);
    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   GaussDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 * 返回值:
 *   BOOL                 - 边缘检测成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用高斯拉普拉斯边缘检测算子对图像进行边缘检测运算。
 *
 * 要求目标图像为灰度图像。
 ************************************************************************/
```

```
BOOL WINAPI GaussDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向缓存图像的指针
    LPSTR   lpDst1;
    LPSTR   lpDst2;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits1;
    HLOCAL      hNewDIBBits1;
    LPSTR   lpNewDIBBits2;
    HLOCAL      hNewDIBBits2;

    // 模板高度
    int         iTempH;

    // 模板宽度
    int         iTempW;

    // 模板系数
    FLOAT   fTempC;

    // 模板中心元素X坐标
    int         iTempMX;

    // 模板中心元素Y坐标
    int         iTempMY;

    //模板数组
    FLOAT aTemplate[25];

    // 暂时分配内存，以保存新图像
    hNewDIBBits1 = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits1 == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits1 = (char * )LocalLock(hNewDIBBits1);

    // 暂时分配内存，以保存新图像
    hNewDIBBits2 = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits2 == NULL)
    {
        // 分配内存失败
        return FALSE;
    }
```

```
// 锁定内存
lpNewDIBBits2 = (char * )LocalLock(hNewDIBBits2);

// 拷贝原图像到缓存图像中
lpDst1 = (char *)lpNewDIBBits1;
memcpy(lpNewDIBBits1, lpDIBBits, lWidth * lHeight);

lpDst2 = (char *)lpNewDIBBits2;
memcpy(lpNewDIBBits2, lpDIBBits, lWidth * lHeight);

// 设置Gauss模板参数
iTempW = 5;
iTempH = 5;
fTempC = 1.0;
iTempMX = 3;
iTempMY = 3;

aTemplate[0] = -2.0;
aTemplate[1] = -4.0;
aTemplate[2] = -4.0;
aTemplate[3] = -4.0;
aTemplate[4] = -2.0;
aTemplate[5] = -4.0;
aTemplate[6] = 0.0;
aTemplate[7] = 8.0;
aTemplate[8] = 0.0;
aTemplate[9] = -4.0;
aTemplate[10] = -4.0;
aTemplate[11] = 8.0;
aTemplate[12] = 24.0;
aTemplate[13] = 8.0;
aTemplate[14] = -4.0;
aTemplate[15] = -4.0;
aTemplate[16] = 0.0;
aTemplate[17] = 8.0;
aTemplate[18] = 0.0;
aTemplate[19] = -4.0;
aTemplate[20] = -2.0;
aTemplate[21] = -4.0;
aTemplate[22] = -4.0;
aTemplate[23] = -4.0;
aTemplate[24] = -2.0;

// 调用Template()函数
if (!Template(lpNewDIBBits1, lWidth, lHeight,
    iTempH, iTempW, iTempMX, iTempMY, aTemplate, fTempC))
{
    return FALSE;
}
```

```
    // 复制经过模板运算后的图像到原图像
    memcpy(lpDIBBits, lpNewDIBBits1, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits1);
    LocalFree(hNewDIBBits1);

    LocalUnlock(hNewDIBBits2);
    LocalFree(hNewDIBBits2);
    // 返回
    return TRUE;
}
```

相应的头文件 edgecontour.h:

```
// edgecontour.h

#define PI 3.1415927
#ifndef _INC_EdgeContourAPI
#define _INC_EdgeContourAPI

// 函数原型

BOOL WINAPI RobertDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI SobelDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI PrewittDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI KirschDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI GaussDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI HoughDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI FillDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI ContourDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI TraceDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight);

#endif //!_INC_EdgeContourAPI

typedef struct{
    int Value;
    int Dist;
    int AngleNumber;
}   MaxValue;

typedef struct{
    int Height;
    int Width;
}   Seed;

typedef struct{
    int Height;
    int Width;
}   Point;
```

下面在在菜单中添加一个“边缘与轮廓”菜单。如图 8－9 所示。

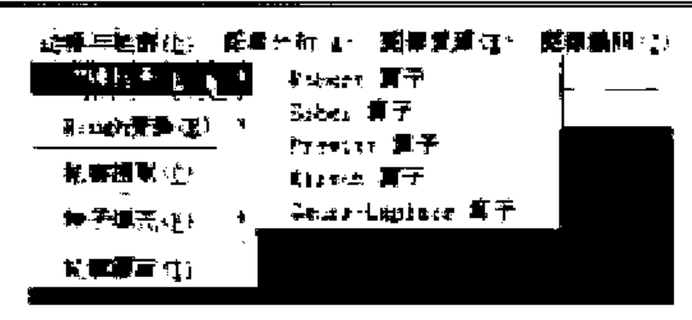


图 8－9 边缘与轮廓菜单

为各个边缘检测的子菜单项添加单击事件处理代码如下：

```
void CCh1_1View::OnEdgeRobert()
{
    //Robert边缘检测运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR  lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用RobertDIB()函数对DIB进行边缘检测
    if (RobertDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
    {
```

```
            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();

    }

    void CCh1_1View::OnEdgeSobel()
    {
        //Sobel边缘检测运算

        // 获取文档
        CCh1_1Doc* pDoc = GetDocument();

        // 指向DIB的指针
        LPSTR   lpDIB;

        // 指向DIB像素指针
        LPSTR     lpDIBBits;

        // 锁定DIB
        lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

        // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
        if (::DIBNumColors(lpDIB) != 256)
        {
            // 提示用户
            MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION |
MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

            // 返回
            return;
        }
```

```
    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用SobelDIB()函数对DIB进行边缘检测
    if (SobelDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();


}
void CCh1_1View::OnEdgePrewitt()
{
    //Prewitt边缘检测运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION |
```

```
MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

            // 返回
            return;
        }

        // 更改光标形状
        BeginWaitCursor();

        // 找到DIB图像像素起始位置
        lpDIBBits = ::FindDIBBits(lpDIB);

        // 调用PrewittDIB()函数对DIB进行边缘检测
        if (PrewittDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
        {

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();

    }
    void CCh1_1View::OnEdgeKirsch()
    {
        //Kirsch边缘检测运算

        // 获取文档
        CCh1_1Doc* pDoc = GetDocument();

        // 指向DIB的指针
        LPSTR    lpDIB;

        // 指向DIB像素指针
        LPSTR     lpDIBBits;
```

```
    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用KirschDIB()函数对DIB进行边缘检测
    if (KirschDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
void CCh1_1View::OnEdgeGauss()
{
    //Gauss边缘检测运算

    // 获取文档
```

```
	CCh1_1Doc* pDoc = GetDocument();

	// 指向DIB的指针
	LPSTR	lpDIB;

	// 指向DIB像素指针
	LPSTR	lpDIBBits;

	// 锁定DIB
	lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

	// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
	if (::DIBNumColors(lpDIB) != 256)
	{
		// 提示用户
		MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION |
MB_OK);

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 返回
		return;
	}

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用GaussDIB()函数对DIB进行边缘检测
	if (GaussDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);

		// 更新视图
		pDoc->UpdateAllViews(NULL);
	}
	else
	{
		// 提示用户
		MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
	}

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
```

```
EndWaitCursor();

}
```

图8－7，图8－8，图8－9所示的处理结果就是用上述程序实现的。

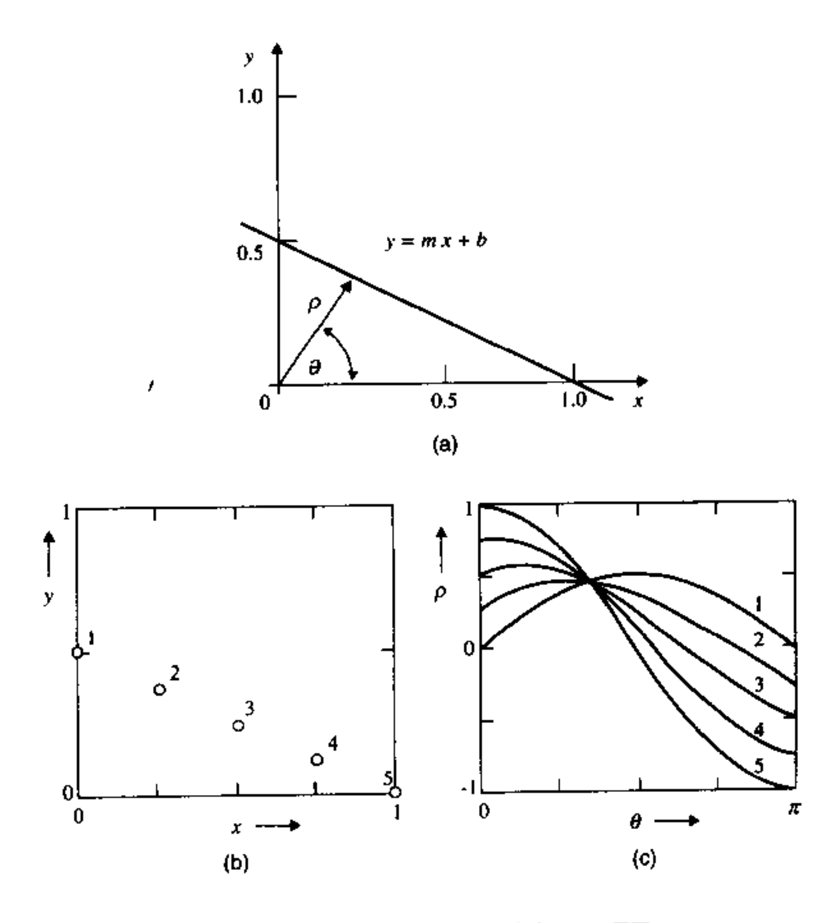
8.2 Hough 变换

8.2.1 基本概念

直线 $y=mx+b$ 可用极坐标表示为：

$r=x\cos(\theta)+y\sin(\theta)$

其中(r, θ)定义了一个从原点到线上最近点的向量(图 8—10a)，这个向量与该直线垂直。



(a)一条直线的极坐标表示，(b)x,y 平面，(c)r, θ平面

图 8－10　Hough 变换

考虑一个以参数 r 和θ定义的二维空间。x,y 平面的任意一直线对应了该空间的一个点。因此，x,y 平面的任意一直线的 Hough 变换是 r, θ空间中的一个点。

现在考虑 x,y 平面的一个特定的点(x_l,y_l)。过该点的直线可以有很多，每一条都对应了 r,θ空间中的一个点。然而这些点必须是满足以 x_l 和 y_l 作为常量时的等式。因此在参数空间中与 x,y 空间中所有这些直线对应的点的轨迹是一条正弦型曲线，而 x,y 平面上的任一点(图 8－10b)对应了 r,θ空间的一条正弦曲线(图 8－10c)。

如果有一组位于由参数 r_0 和θ_0 决定的直线上的边缘点，则每个边缘点对应了 r,θ空间的一条正弦型曲线。所有这些曲线必交于点(r_0,θ_0)，因为这是它们共享的一条直线的参数(图 8－10c)。

为了找出这些点所构成的直线段，我们可以将 r,θ空间量化成许多小格。根据每一个(x_0,y_0)点代入θ的量化值，算出各个 r，所得值(经量化)落在某个小格内，便使该小格的计数累加器加 1，当全部(x,y)点变换后，对小格进行检验，有大的计数值的小格对应于共线点，其(r,θ)可用作直线拟合参数。有小的计数值的各小格一般反映非共线点，应丢弃不用。

可以看出，如果 r 和θ量化得过粗，则参数空间的凝聚效果较差，找不出直线的准确的 r 和θ值；反过来，如果 r,θ量化得过细，那么计算量将增大，需要兼顾这两方面，取合适的量化值。

若图像中各点是边沿元，而且梯度方向已求出，在寻找有无直线边沿时可在其梯度方向内把θ精细量化，其他θ角则粗量化，这样在不增加总的量化小格数的情况下，可以提高检测直线边沿的方向角的精度。

对于圆，可写出其方程：

$$(x-a)^2+(y-b)^2=R^2$$

这时参数空间增加到三维，由 a,b,R 组成。如果仍然像找直线那样直接计算，那么计算量和存储空间都将显著增大。

如果已知有圆的边沿元，而且边沿元为已知，那么可以简化为二维的问题。因为把上式对 x 取导数，有：

$$2(x-a)+2(y-b)\frac{\mathrm{d}y}{\mathrm{d}x}=0$$

这表示参数 a 和 b 不独立，利用这个关系以后，解上式只需用二个参数(例如 b 和 R)组成参数空间，计算量减少了很多。

在人为景物中圆形物体经常出现，经过透视成像后由圆变成椭圆。寻找椭圆的算法可以仿照寻找圆的算法来进行。

设椭圆方程为：

$$\frac{(x-x_0)^2}{a^2}+\frac{(y-y_0)^2}{b^2}=1$$

取导数有：

$$\frac{(x-x_0)}{a^2}+\frac{(y-y_0)}{b^2}\frac{\mathrm{d}y}{\mathrm{d}x}=0$$

可以看到这里有三个独立参数。如果椭圆主轴不平行于坐标轴，则可写为

$$Ax^2 + Bxy + Cy^2 + Dx + Ey + 1 = 0$$

在利用椭圆边沿的方向信息后，在映射空间的独立参数仍有四个之多，为了简化求椭圆的计算，还需要其他的特殊解法，这里就不多介绍了。

8.2.2 Visual C++编程实现

下面我们介绍一个利用 Hough 变换来检测图像中是否存在平行直线的程序。程序的计算步骤如下：

(1) 初始化一个变换域 r,θ空间的数组，r 方向上的量化数目图像对角线方向像素数，θ 方向上的量化数目为 90（角度从 0～180，每格 2 度）。

(2) 顺序搜索图像中的所有黑点。对每一个黑点，在变换域的对应各点上加 1。

(3) 求出变换域中的最大值点并记录。

(4) 将最大值点及其附近的点清零。

(5) 求出变换域中的第二个最大值点并记录。

(6) 判断这两个最大值点是否对应两条平行的直线。如果是，则画出这两条平行直线；否则结束。

在程序中用到了一个自己在 edgecontour.h 中定义的数据结构 MaxValue:

```
typedef struct{
    int Value;
    int Dist;
    int AngleNumber;
}   MaxValue;
```

实现这一功能的函数为 HoughDIB()。

```
/*************************************************************************
 *
 * 函数名称:
 *   HoughDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 * 返回值:
 *   BOOL                 - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对检测图像中的平行直线。如果图像中有两条平行的直线，则将这两条平行直线
 * 提取出来。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI HoughDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
```

```
LPSTR   lpSrc;

// 指向缓存图像的指针
LPSTR   lpDst;

// 指向变换域的指针
LPSTR     lpTrans;

// 图像每行的字节数
LONG lLineBytes;

// 指向缓存DIB图像的指针
LPSTR   lpNewDIBBits;
HLOCAL      hNewDIBBits;

//指向变换域的指针
LPSTR   lpTransArea;
HLOCAL      hTransArea;

//变换域的尺寸
int iMaxDist;
int iMaxAngleNumber;

//变换域的坐标
int iDist;
int iAngleNumber;

//循环变量
long i;
long j;

//像素值
unsigned char pixel;

//存储变换域中的两个最大值
MaxValue MaxValue1;
MaxValue MaxValue2;

// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化新分配的内存，设定初始值为255
```

```
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lWidth * lHeight);

//计算变换域的尺寸
//最大距离
iMaxDist = (int) sqrt(lWidth*lWidth + lHeight*lHeight);

//角度从0－180，每格2度
iMaxAngleNumber = 90;

//为变换域分配内存
hTransArea = LocalAlloc(LHND, lWidth * lHeight * sizeof(int));

if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpTransArea = (char * )LocalLock(hTransArea);

// 初始化新分配的内存，设定初始值为0
lpTrans = (char *)lpTransArea;
memset(lpTrans, 0, lWidth * lHeight * sizeof(int));

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

for(j = 0; j <lHeight; j++)
{
    for(i = 0;i <lWidth; i++)
    {

        // 指向原图像倒数第j行，第i个像素的指针
        lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

        //取得当前指针处的像素值，注意要转换为unsigned char型
        pixel = (unsigned char)*lpSrc;

        //目标图像中含有0和255外的其他灰度值
        if(pixel != 255 && *lpSrc != 0)
            return FALSE;

        //如果是黑点，则在变换域的对应各点上加1
        if(pixel == 0)
        {
            //注意步长是2度
            for(iAngleNumber=0; iAngleNumber<iMaxAngleNumber; iAngleNumber++)
            {
                iDist = (int) fabs(i*cos(iAngleNumber*2*PI/180.0) + \
```

```
                        j*sin(iAngleNumber*2*PI/180.0));

                    //变换域的对应点上加1
                    *(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber) = \
                        *(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber) +1;
                }
            }

        }
    }

    //找到变换域中的两个最大值点
    MaxValue1.Value=0;
    MaxValue2.Value=0;

    //找到第一个最大值点
    for (iDist=0; iDist<iMaxDist;iDist++)
    {
        for(iAngleNumber=0; iAngleNumber<iMaxAngleNumber; iAngleNumber++)
        {
            if((int)*(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber)>MaxValue1.Value)
            {
                MaxValue1.Value = (int)*(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber);
                MaxValue1.Dist = iDist;
                MaxValue1.AngleNumber = iAngleNumber;
            }

        }
    }

    //将第一个最大值点附近清零
    for (iDist = -9;iDist < 10;iDist++)
    {
        for(iAngleNumber=-1; iAngleNumber<2; iAngleNumber++)
        {
            if(iDist+MaxValue1.Dist>=0 && iDist+MaxValue1.Dist<iMaxDist \
                && iAngleNumber+MaxValue1.AngleNumber>=0 &&
iAngleNumber+MaxValue1.AngleNumber<=iMaxAngleNumber)
            {
                *(lpTransArea+(iDist+MaxValue1.Dist)*iMaxAngleNumber+\
                    (iAngleNumber+MaxValue1.AngleNumber))=0;
            }
        }
    }

    //找到第二个最大值点
    for (iDist=0; iDist<iMaxDist;iDist++)
    {
        for(iAngleNumber=0; iAngleNumber<iMaxAngleNumber; iAngleNumber++)
        {
            if((int)*(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber)>MaxValue2.Value)
```

```
                {
                    MaxValue2.Value = (int)*(lpTransArea+iDist*iMaxAngleNumber+iAngleNumber);
                    MaxValue2.Dist = iDist;
                    MaxValue2.AngleNumber = iAngleNumber;
                }

        }
    }


    //判断两直线是否平行
    if(abs(MaxValue1.AngleNumber-MaxValue2.AngleNumber)<=2)
    {
        //两直线平行，在缓存图像中重绘这两条直线
        for(j = 0; j <lHeight; j++)
        {
            for(i = 0;i <lWidth; i++)
            {

                // 指向缓存图像倒数第j行，第i个像素的指针
                lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;

                //如果该点在某一条平行直线上，则在缓存图像上将该点赋为黑

                //在第一条直线上
                iDist = (int) fabs(i*cos(MaxValue1.AngleNumber*2*PI/180.0) + \
                            j*sin(MaxValue1.AngleNumber*2*PI/180.0));
                if (iDist == MaxValue1.Dist)
                    *lpDst = (unsigned char)0;

                //在第二条直线上
                iDist = (int) fabs(i*cos(MaxValue2.AngleNumber*2*PI/180.0) + \
                            j*sin(MaxValue2.AngleNumber*2*PI/180.0));
                if (iDist == MaxValue2.Dist)
                    *lpDst = (unsigned char)0;
            }
        }
    }



    // 复制腐蚀后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 释放内存
    LocalUnlock(hTransArea);
```

```
		LocalFree(hTransArea);

		// 返回
		return TRUE;

	}
```

对应的菜单单击事件处理程序如下：

```
void CCh1_1View::OnEdgeHough()
{
	//Hough运算

	// 获取文档
	CCh1_1Doc* pDoc = GetDocument();

	// 指向DIB的指针
	LPSTR	lpDIB;

	// 指向DIB像素指针
	LPSTR	lpDIBBits;

	// 锁定DIB
	lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

	// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
	if (::DIBNumColors(lpDIB) != 256)
	{
		// 提示用户
		MessageBox("目前只支持256色位图的运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 返回
		return;
	}

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用HoughDIB()函数对DIB
	if (HoughDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);

		// 更新视图
```

```
			pDoc->UpdateAllViews(NULL);
		}
		else
		{
			// 提示用户
			MessageBox("分配内存失败或者图像中含有0和255之外的像素值！", "系统提示",
MB_ICONINFORMATION | MB_OK);
		}

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 恢复光标
		EndWaitCursor();

	}
```

图 8－11 和图 8－12 给出了程序的运行结果。

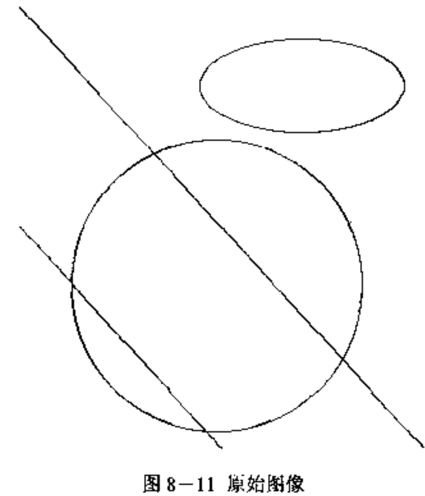


图 8－11 原始图像

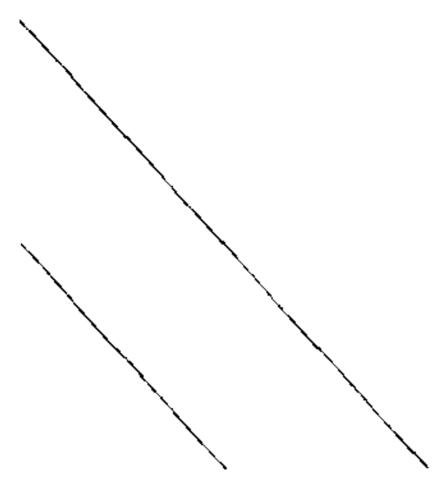


图 8－12 检测出的平行直线

8.3 轮廓提取与轮廓跟踪

8.3.1 基本概念

轮廓提取和轮廓跟踪的目的都是获得图像的外部轮廓特征。在必要的情况下应用一定的方法表达轮廓的特征，为图像的形状分析做准备。

二值图像轮廓提取的算法非常简单，就是掏空内部点：如果原图中有一点为黑，且它的8个相邻点都是黑色时（此时该点是内部点），则将该点删除。

联系到上一章数学形态学的内容，可以看到，这实际上相当于用一个九个点的结构元素对原图像进行腐蚀，再用原图像减去腐蚀图像。

图像的轮廓提取过程如图8－13及图8－14所示。

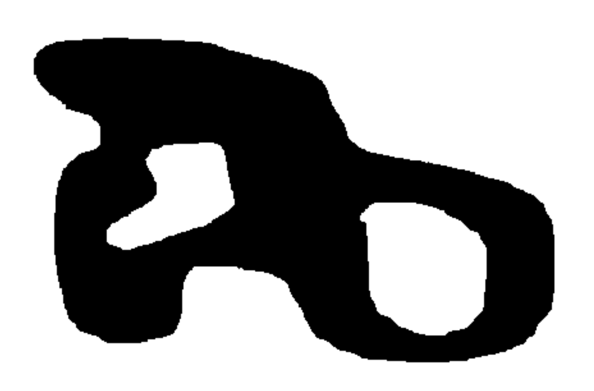


图8－13 原始图像

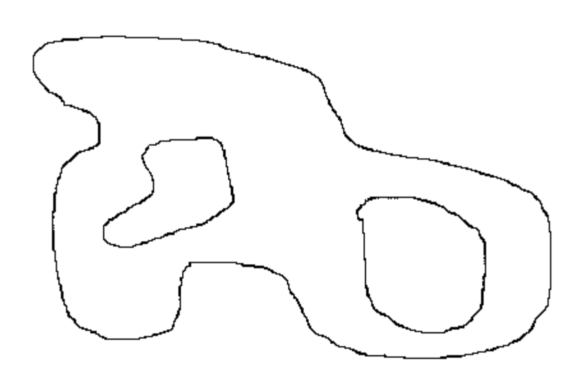


图8－14 轮廓提取的结果

轮廓跟踪的基本方法是：先根据某些严格的“探测准则”找出目标物体轮廓上的像素，再根据这些像素的某些特征用一定的“跟踪准则”找出目标物体上的其他像素。下面来介绍两种二值图像轮廓跟踪的算法。

首先找到第一个边界像素的“探测准则”是：按照从左到右，从下到上的顺序搜索，找到的第一个黑点一定是最左下方的边界点，记为 *A*。它的右、右上、上、左上四个邻点中至少有一个是边界点，记为 *B*。从 *B* 开始找起，按右、右上、上、左上、左、左下、下、右下的顺序找相邻点中的边界点 *C*。如果 *C* 就是 *A* 点，则表明已经转了一圈，程序结束；否则从 *C* 点继续找，直到找到 *A* 为止。 判断是不是边界点很容易：如果它的上下左右四个邻点都不是黑点则它即为边界点。（即跟踪准则）。

这种算法要对每个边界像素周围的八个点进行判断，计算量比较大。下面我们来看另外一种跟踪准则。

首先按照上面所说的“探测准则”找到最左下方的边界点。以这个边界点起始，假设已经沿顺时针方向环绕整个图像一圈找到了所有的边界点。由于边界是连续的，所以每一个边界点都可以用这个边界点对前一个边界点所张的角度来表示。因此可以使用下面的跟踪准则：从第一个边界点开始，定义初始的搜索方向为沿左上方；如果左上方的点是黑点，则为边界点，否则搜索方向顺时针旋转 45 度。这样一直到找到第一个黑点为止。然后把这个黑点作为新的边界点，在当前搜索方向的基础上逆时针旋转 90 度，继续用同样的方法继续搜索下一个黑点，直到返回最初的边界点为止。

图 8－15 为这一轮廓跟踪算法的示意图，其中箭头代表搜索方向。

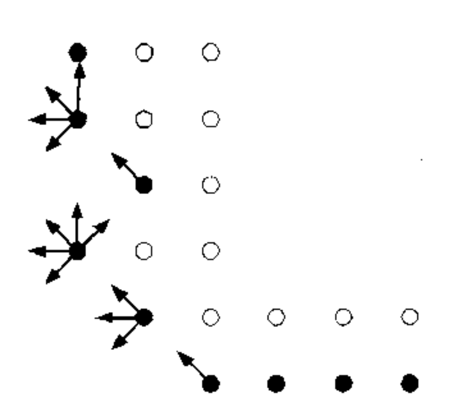


图 8－15 轮廓跟踪算法

图 8－16 给出了使用这一算法对图 8－13 进行轮廓跟踪的结果。

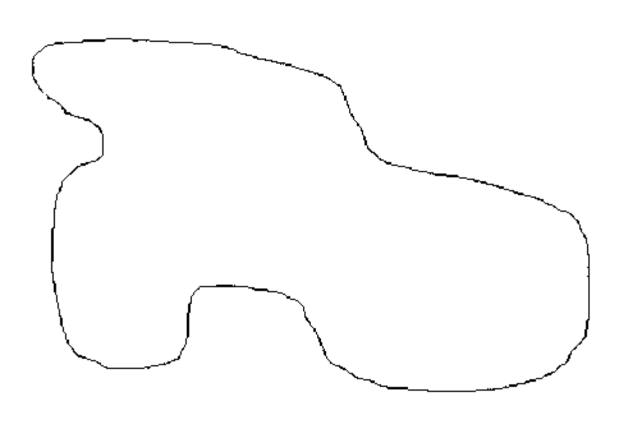


图 8－16 轮廓跟踪的结果

8.3.2 Visual C++编程实现

我们下面来看一下实现轮廓提取和轮廓跟踪的 ContourDIB()和 TraceDIB()函数：

```
/*************************************************************************
 *
 * 函数名称:
 *   ContourDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 * 返回值:
 *   BOOL                 - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行轮廓提取运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI ContourDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
```

```
    LPSTR   lpNewDIBBits;
    HLOCAL      hNewDIBBits;

    //循环变量
    long i;
    long j;
    unsigned char n,e,s,w,ne,se,nw,sw;

    //像素值
    unsigned char pixel;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);
    for(j = 1; j <lHeight-1; j++)
    {
        for(i = 1;i <lWidth-1; i++)
        {

            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lWidth * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lWidth * j + i;

            //取得当前指针处的像素值，注意要转换为unsigned char型
            pixel = (unsigned char)*lpSrc;

            //目标图像中含有0和255外的其他灰度值
//          if(pixel != 255 && pixel != 0)
//              return FALSE;
            if(pixel == 0)
            {
                *lpDst = (unsigned char)0;
                nw = (unsigned char)*(lpSrc + lWidth -1);
                n   = (unsigned char)*(lpSrc + lWidth );
                ne = (unsigned char)*(lpSrc + lWidth +1);
                w = (unsigned char)*(lpSrc -1);
                e = (unsigned char)*(lpSrc +1);
```

```
                sw = (unsigned char)*(lpSrc - lWidth -1);
                s  = (unsigned char)*(lpSrc - lWidth );
                se = (unsigned char)*(lpSrc - lWidth +1);
                //如果相邻的八个点都是黑点
                if(nw+n+ne+w+e+sw+s+se==0)
                {
                    *lpDst = (unsigned char)255;
                }
            }
        }
    }

    // 复制运算后的图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   TraceDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LONG  lWidth        - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight       - 原图像高度（像素数）
 * 返回值:
 *   BOOL                - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行轮廓跟踪运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/
BOOL WINAPI TraceDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
```

```
HLOCAL      hNewDIBBits;

// 图像每行的字节数
LONG lLineBytes;

//循环变量
long i;
long j;

//像素值
unsigned char pixel;

//是否找到起始点及回到起始点
bool bFindStartPoint;

//是否扫描到一个边界点
bool bFindPoint;

//起始边界点与当前边界点
Point StartPoint,CurrentPoint;

//八个方向和起始扫描方向
int Direction[8][2]={{-1,1},{0,1},{1,1},{1,0},{1,-1},{0,-1},{-1,-1},{-1,0}};
int BeginDirect;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lLineBytes * lHeight);

if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化新分配的内存，设定初始值为255
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lLineBytes * lHeight);

//先找到最左上方的边界点
bFindStartPoint = false;
for (j = 0;j < lHeight && !bFindStartPoint;j++)
{
    for(i = 0;i < lWidth && !bFindStartPoint;i++)
    {
        // 指向原图像倒数第j行，第i个像素的指针
```

```
                lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                if(pixel == 0)
                {
                    bFindStartPoint = true;

                    StartPoint.Height = j;
                    StartPoint.Width = i;

                    // 指向目标图像倒数第j行，第i个像素的指针
                    lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;
                    *lpDst = (unsigned char)0;
                }
            }
        }

        //由于起始点是在左下方，故起始扫描沿左上方向
        BeginDirect = 0;
        //跟踪边界
        bFindStartPoint = false;
        //从初始点开始扫描
        CurrentPoint.Height = StartPoint.Height;
        CurrentPoint.Width = StartPoint.Width;
        while(!bFindStartPoint)
        {
            bFindPoint = false;
            while(!bFindPoint)
            {
                //沿扫描方向查看一个像素
                lpSrc = (char *)lpDIBBits + lLineBytes * ( CurrentPoint.Height + Direction[BeginDirect][1])
                    + (CurrentPoint.Width + Direction[BeginDirect][0]);
                pixel = (unsigned char)*lpSrc;
                if(pixel == 0)
                {
                    bFindPoint = true;
                    CurrentPoint.Height = CurrentPoint.Height + Direction[BeginDirect][1];
                    CurrentPoint.Width = CurrentPoint.Width + Direction[BeginDirect][0];
                    if(CurrentPoint.Height == StartPoint.Height && CurrentPoint.Width ==
StartPoint.Width)
                    {
                        bFindStartPoint = true;
                    }
                    lpDst = (char *)lpNewDIBBits + lLineBytes * CurrentPoint.Height + CurrentPoint.Width;
                    *lpDst = (unsigned char)0;
                    //扫描的方向逆时针旋转两格
                    BeginDirect--;
                    if(BeginDirect == -1)
                        BeginDirect = 7;
```

```
                    BeginDirect--;
                    if(BeginDirect == -1)
                        BeginDirect = 7;
                }
                else
                {
                    //扫描方向顺时针旋转一格
                    BeginDirect++;
                    if(BeginDirect == 8)
                        BeginDirect = 0;

                }
            }
        }

        // 复制运算后的图像
        memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

        // 释放内存
        LocalUnlock(hNewDIBBits);
        LocalFree(hNewDIBBits);

        // 返回
        return TRUE;
}
```

对应的菜单单击事件处理代码如下：

```
void CCh1_1View::OnEdgeContour()
{
    //轮廓提取运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
```

```
        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用ContourDIB()函数对DIB进行轮廓提取
    if (ContourDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
}

void CCh1_1View::OnEdgeTrace()
{
    //轮廓跟踪运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
```

```
		if (::DIBNumColors(lpDIB) != 256)
		{
			// 提示用户
			MessageBox("目前只支持256色位图的运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

			// 解除锁定
			::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

			// 返回
			return;
		}

		// 更改光标形状
		BeginWaitCursor();

		// 找到DIB图像像素起始位置
		lpDIBBits = ::FindDIBBits(lpDIB);

		// 调用TraceDIB()函数对DIB进行轮廓跟踪
		if (TraceDIB(lpDIBBits, ::DIBWidth(lpDIB) , ::DIBHeight(lpDIB)))
		{

			// 设置脏标记
			pDoc->SetModifiedFlag(TRUE);

			// 更新视图
			pDoc->UpdateAllViews(NULL);
		}
		else
		{
			// 提示用户
			MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
		}

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 恢复光标
		EndWaitCursor();

	}
```

8.4 种子填充

8.4.1 基本概念

种子填充算法是图形学中的算法，是轮廓提取算法的逆过程。

种子填充算法首先假定封闭轮廓线内某点是已知的，然后算法开始搜索与种子点相邻且位于轮廓线内的点。如果相邻点不在轮廓线内，那么就到达轮廓线的边界；如果相邻点位于轮廓线之内，那么这一点就成为新的种子点，然后继续地搜索下去。种子填充区域的连通情况又有四连通和八连通之分：

● 四连通区域

各像素在水平和垂直四个方向上是连通的。如图 8－17 所示，其中 (a)和(b)是四连通式内部定义的区域；而(c)和(d)是四连通式边界定义的区域。

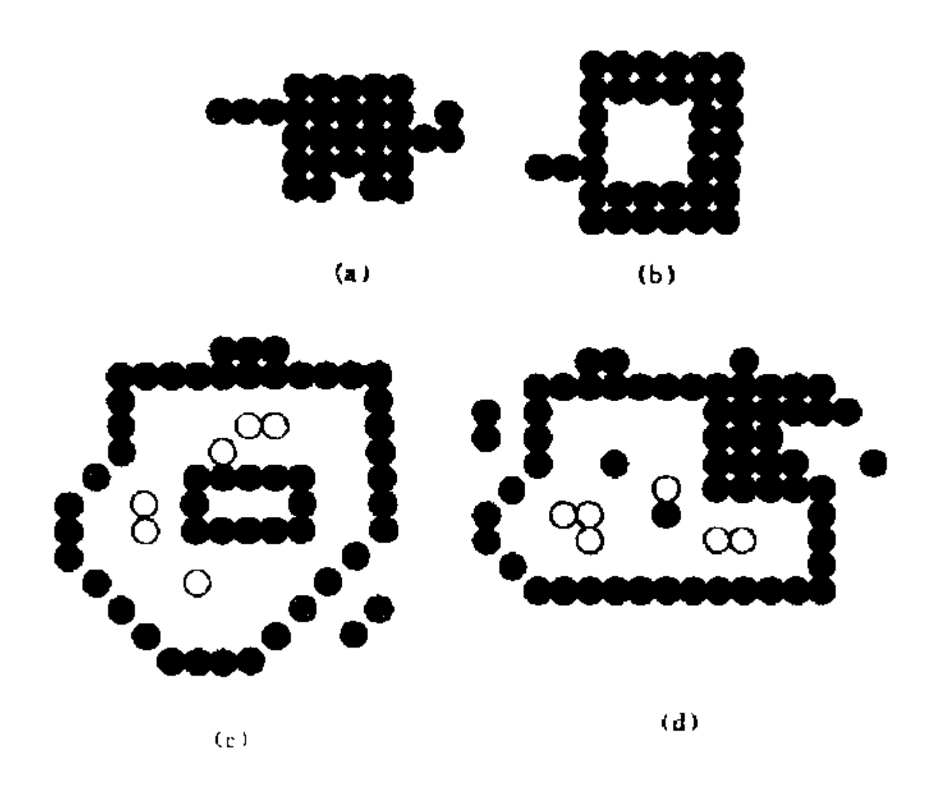
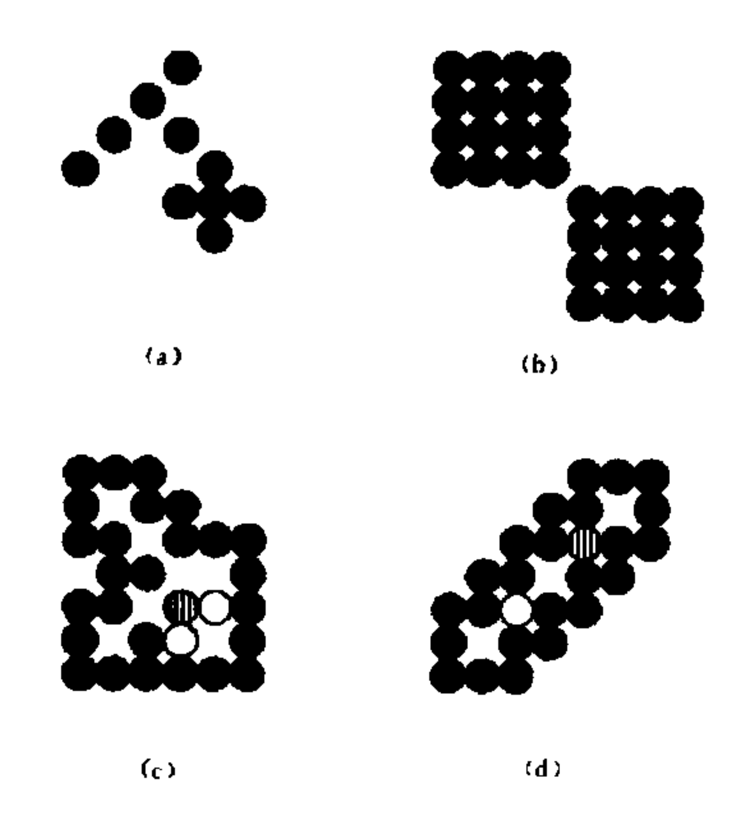


图 8－17 四连通式区域

● 八连通区域

各像素在水平、垂直及四个对角线方向都是连通的。如图 8－18 所示，其中(a)及(b)是八连通式内部定义的区域，而 (c)和(d)则是八连通式边界定义的区域。



(a) (b) (c) (d)

图 8－18 八连通式区域

一个八连通式区域的边界是四连通式的，而一个四连通式区域的边界则是八连通式的。因此，一个八连通式区域的算法可以用在四连通式的区域上，但是由于它可以“跳过”像素之间的对角线连线，故有可能越界而产生意想不到的结果。

最简单的种子填充算法称为漫水法。这是对内定义区域进行填充的算法，这一算法所采用的基本方法是：首先在区域内测试一点(x,y)的像素值，看其是否具有原始给定的值，也就是决定该点是否在区域内未被填充过。如果是，则改变其颜色或亮度值，然后再在其四个方向或八个方向上扩展，继续测试，通过反复调用，实现四连通式或八连通式的区域填充。

边界填充算法与漫水法的基本思想是一样的，所不同的是，在测试(x, y)点的像素是否处在区域之内且又未被访问过时，包括两部分的内容：①与边界值相比较，以检测此像素是否为该区域的一部分；②与新值相比较，以检测该像素是否已被访问过。这种测试的前提条件是：在初始状态，区域内没有一个像素已设置为新值，但允许新值等于边界值。

我们可以用堆栈的方法，对边界定义的区域进行填充。基本流程是：

(1) 种子像素压入堆栈；

(2) 当堆栈非空时，从堆栈中推出一个像素，并将该像素设置成所要的值；

(3) 对于每个与当前像素邻接的四连通或八连通像素，进行上述两部分内容的测试；

(4) 若所测试的像素在区域内没有被填充过，则将该像素压入堆栈。

上述种子填充算法过程虽然很简单，但却是深度递归的。递归要花费时间，当内存空间

有限时，还可能引起栈溢出。图 8－19 所表示的是由顶点(1,1)、(8,1)、(8,4)、(6,6)及(1,6)所决定的边界定义多边形区域，种子像素为(4,3)。当用堆栈方法对其进行填充时，我们发现，有些像素多次被压入堆栈，使堆栈变得很大。如果按照右、上、左、下四连通域填充，当算法进行到像素(5,5)时，堆栈的深度为 23 层，栈内包含很多重复的和不必要的信息。

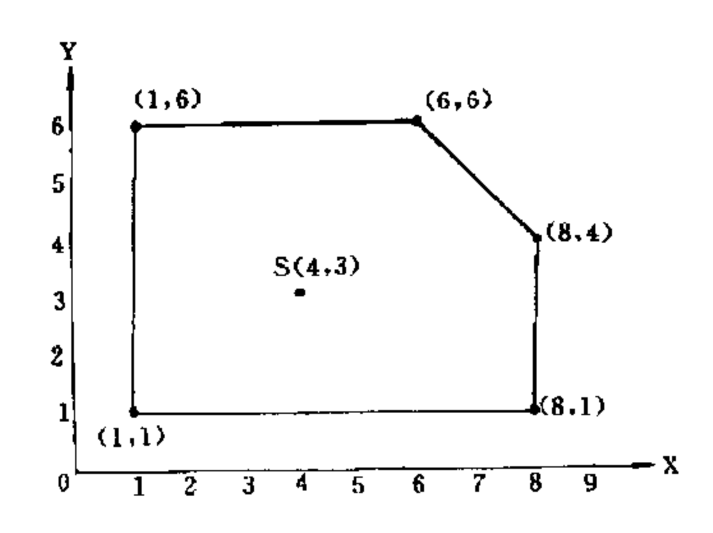


图 8－19 填充边界定义的区域

使堆栈极小化的一种算法是：在任意不间断的扫描线区段中，只取一个种子像素，称为扫描线种子填充算法。

上述简单种子填充算法效率低的主要原因是我们并未考虑像素间的相关性，而孤立的对单个像素进行测试。扫描线种子填充算法的测试对象是一个个像素段。这里，像素段是指区域内相邻像素在水平方向的组合，它的两端以边界值的像素为边界，其中不包括具有新值的像素。对于区域内的每一像素段，我们可以只保留其最右(或左)端的像素作为种子像素。因此，区域中每一个未被填充的部分，至少有一个像素段是保持在栈里的。扫描线种子填充算法适用于边界定义的区域。区域可以是凸的，也可以是凹的，还可以包含一个或多个孔，如图 8－20 所示。

这个算法按下述步骤进行：

(1) 从包含种子像素的堆栈中推出区段的种子像素；

(2) 沿着扫描线，对种子像素的左、右像素进行填充，直至遇到边界像素为止；

(3) 区段内最左和最右的像素记为 X_l 和 X_r。在 $X_l \leq X \leq X_r$ 时，检查与当前扫描线相邻的上、下两条扫描线是否全为边界像素或已填充过；

(4) 如果这些扫描线既不包含边界像素，也不包含已填充的像素，那么把每一像素段的最右像素取作为种子像素，并压入堆栈；

(5) 初始化时，向堆栈压入一个种子像素，并在堆栈为空时结束。

用这一算法填充图 8－19 所示的区域，堆栈深度最大为 2，从而解决了简单种子填充算法存在的问题。

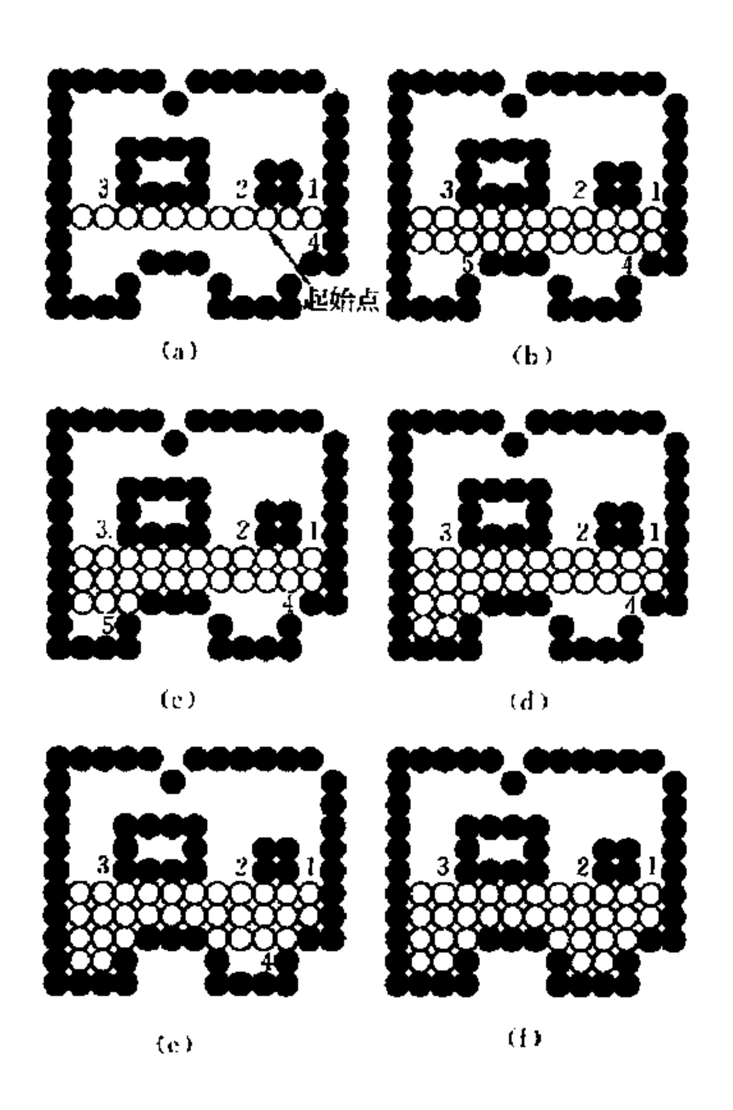


图 8－20 用扫描线种子填充算法填充边界定义的区域

8.4.2 Visual C++编程实现

下面我们分别实现简单的种子填充算法以及扫描线种子填充算法。在简单的种子填充算法中，首先初始化一个堆栈，并以图像的中心点作为种子，先将要填充的点 push 进堆栈中，以后每 pop 出一个点都将该点涂成黑色，然后按左上右下的顺序查看它的四个相邻点，若为白（表示还没有填充），则将该邻点 push 进栈。如此循环，直到堆栈为空。此时，区域内所有的点都被涂成了黑色。

简单的种子填充算法由函数 FillDIB()实现。其中定义了一个堆栈数组和堆栈指针：

```
Seeds = new Seed[lWidth*lHeight];
```

Seed 是在 edgecontour.h 中定义的结构：

```
typedef struct{
    int Height;
    int Width;
}   Seed;

/*************************************************************************
 *
 * 函数名称：
 *   FillDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG   lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG   lHeight        - 原图像高度（像素数）
 * 返回值:
 *   BOOL                   - 种子填充成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行种子填充运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI FillDIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    //像素值
    unsigned char pixel;

    //种子堆栈及指针
    Seed *Seeds;
    int StackPoint;

    //当前像素位置
    int iCurrentPixelx,iCurrentPixely;

    //初始化种子
    Seeds = new Seed[lWidth*lHeight];
    Seeds[1].Height = lHeight / 2;
    Seeds[1].Width = lWidth / 2;
    StackPoint = 1;

    while( StackPoint != 0)
    {
        //取出种子
        iCurrentPixelx = Seeds[StackPoint].Width;
        iCurrentPixely = Seeds[StackPoint].Height;
        StackPoint--;
```

```
lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + iCurrentPixelx;
//取得当前指针处的像素值，注意要转换为unsigned char型
pixel = (unsigned char)*lpSrc;

//目标图像中含有0和255外的其他灰度值
if(pixel != 255 && pixel != 0)
    return FALSE;

//将当前点涂黑
*lpSrc = (unsigned char)0;

//判断左面的点，如果为白，则压入堆栈
//注意防止越界
if(iCurrentPixelx > 0)
{
    lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + iCurrentPixelx - 1;
    //取得当前指针处的像素值，注意要转换为unsigned char型
    pixel = (unsigned char)*lpSrc;
    if (pixel == 255)
    {
        StackPoint++;
        Seeds[StackPoint].Height = iCurrentPixely;
        Seeds[StackPoint].Width = iCurrentPixelx - 1;
    }
    //目标图像中含有0和255外的其他灰度值
    if(pixel != 255 && pixel != 0)
        return FALSE;
}

//判断上面的点，如果为白，则压入堆栈
//注意防止越界
if(iCurrentPixely < lHeight - 1)
{
    lpSrc = (char *)lpDIBBits + lWidth * (iCurrentPixely + 1) + iCurrentPixelx;
    //取得当前指针处的像素值，注意要转换为unsigned char型
    pixel = (unsigned char)*lpSrc;
    if (pixel == 255)
    {
        StackPoint++;
        Seeds[StackPoint].Height = iCurrentPixely + 1;
        Seeds[StackPoint].Width = iCurrentPixelx;
    }
    //目标图像中含有0和255外的其他灰度值
    if(pixel != 255 && pixel != 0)
        return FALSE;
}

//判断右面的点，如果为白，则压入堆栈
//注意防止越界
if(iCurrentPixelx < lWidth - 1)
```

```
		{
			lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + iCurrentPixelx + 1;
			//取得当前指针处的像素值，注意要转换为unsigned char型
			pixel = (unsigned char)*lpSrc;
			if (pixel == 255)
			{
				StackPoint++;
				Seeds[StackPoint].Height = iCurrentPixely;
				Seeds[StackPoint].Width = iCurrentPixelx + 1;
			}
			//目标图像中含有0和255外的其他灰度值
			if(pixel != 255 && pixel != 0)
				return FALSE;
		}

		//判断下面的点，如果为白，则压入堆栈
		//注意防止越界
		if(iCurrentPixely > 0)
		{
			lpSrc = (char *)lpDIBBits + lWidth * (iCurrentPixely - 1) + iCurrentPixelx;
			//取得当前指针处的像素值，注意要转换为unsigned char型
			pixel = (unsigned char)*lpSrc;
			if (pixel == 255)
			{
				StackPoint++;
				Seeds[StackPoint].Height = iCurrentPixely - 1;
				Seeds[StackPoint].Width = iCurrentPixelx;
			}
			//目标图像中含有0和255外的其他灰度值
			if(pixel != 255 && pixel != 0)
				return FALSE;
		}
	}

	//释放堆栈
	delete Seeds;
	// 返回
	return TRUE;
}
```

相应的菜单事件处理代码如下：

```
void CCh1_1View::OnEdgeFill()
{
	//种子填充运算

	// 获取文档
	CCh1_1Doc* pDoc = GetDocument();

	// 指向DIB的指针
	LPSTR	lpDIB;

	// 指向DIB像素指针
```

```
        LPSTR    lpDIBBits;

        // 锁定DIB
        lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

        // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的种子填充，其他的可以类推）
        if (::DIBNumColors(lpDIB) != 256)
        {
            // 提示用户
            MessageBox("目前只支持256色位图的运算！", "系统提示" , MB_ICONINFORMATION |
MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

            // 返回
            return;
        }

        // 更改光标形状
        BeginWaitCursor();

        // 找到DIB图像像素起始位置
        lpDIBBits = ::FindDIBBits(lpDIB);

        // 调用FillDIB()函数对DIB进行种子填充
        if (FillDIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
        {

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();

    }
```

下面给出一个扫描线种子填充算法的程序。算法由函数 edgecontour.cpp 中的 Fill2DIB() 实现，其中定义了一个最大深度为 10 的堆栈数组 Seeds[10]和堆栈指针 StackPoint。在

edgecontour.h 中还定义了数据结构 Point:

```
typedef struct{
    int Height;
    int Width;
}   Point;
```

初始的种子仍然设定为图像的中心点。

Fill2DIB()程序如下:

```
/*************************************************************************
 *
 * 函数名称:
 *   Fill2DIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 * 返回值:
 *   BOOL                 - 种子填充成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行种子填充运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/
BOOL WINAPI Fill2DIB(LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;

    //像素值
    unsigned char pixel;

    //左右边界像素位置
    int xl,xr;

    //是否已填充至边界
    BOOL bFilll,bFillr;

    //种子堆栈及指针
    Seed Seeds[10];
    int StackPoint;

    //当前像素位置
    int iCurrentPixelx,iCurrentPixely;
    int iBufferPixelx,iBufferPixely;
```

```
//初始化种子
Seeds[1].Height = lHeight / 2;
Seeds[1].Width = lWidth / 2;
StackPoint = 1;

while( StackPoint != 0)
{
    //取出种子
    iCurrentPixelx = Seeds[StackPoint].Width;
    iCurrentPixely = Seeds[StackPoint].Height;
    StackPoint--;
    bFilll = true;
    bFillr = true;
    //填充种子所在的行
    //保存种子像素的位置
    iBufferPixelx = iCurrentPixelx;
    iBufferPixely = iCurrentPixely;
    //先向左填充
    while(bFilll)
    {
        lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + iCurrentPixelx;
        //取得当前指针处的像素值，注意要转换为unsigned char型
        pixel = (unsigned char)*lpSrc;

        //目标图像中含有0和255外的其他灰度值
        if(pixel != 255 && pixel != 0)
            return FALSE;
        //遇到边界
        if(pixel == 0)
        {
            bFilll = false;
            xl=iCurrentPixelx+1;
        }
        else
        {
            *lpSrc = (unsigned char)0;
            iCurrentPixelx--;
            //防止越界
            if(iCurrentPixelx<0)
            {
                bFilll = false;
                iCurrentPixelx = 0;
                xl = 0;
            }
        }
    }
    //再向右填充
    //取回种子像素的位置
    iCurrentPixelx = iBufferPixelx+1;
    if(iCurrentPixelx>lWidth)
    {
```

```
        bFillr = false;
        iCurrentPixelx = lWidth;
        xr = lWidth;
    }
    iCurrentPixely = iBufferPixely;
    while(bFillr)
    {
        lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + iCurrentPixelx;
        //取得当前指针处的像素值，注意要转换为unsigned char型
        pixel = (unsigned char)*lpSrc;

        //目标图像中含有0和255外的其他灰度值
        if(pixel != 255 && pixel != 0)
            return FALSE;
        //遇到边界
        if(pixel == 0)
        {
            bFillr = false;
            xr=iCurrentPixelx-1;
        }
        else
        {
            *lpSrc = (unsigned char)0;
            iCurrentPixelx++;
            //防止越界
            if(iCurrentPixelx>lWidth)
            {
                bFillr = false;
                iCurrentPixelx = lWidth;
                xr = lWidth;
            }
        }
    }
    //上、下两条扫描线是否全为边界像素或已填充过
    //先看上面的扫描线
    iCurrentPixely--;
    if(iCurrentPixely < 0)
    {
        iCurrentPixely = 0;
    }
    for (i = xr; i>= xl;i--)
    {
        lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + i;
        //取得当前指针处的像素值，注意要转换为unsigned char型
        pixel = (unsigned char)*lpSrc;

        //有未填充的像素，将新的种子压入堆栈
        if (pixel == 255)
        {
            StackPoint++;
            Seeds[StackPoint].Height = iCurrentPixely;
```

```
                    Seeds[StackPoint].Width = i;
                    break;
                }
            }
            //再看下面的扫描线
            iCurrentPixely+=2;
            if(iCurrentPixely > lHeight)
            {
                iCurrentPixely = lHeight;
            }
            for (i = xr; i>= xl;i--)
            {
                lpSrc = (char *)lpDIBBits + lWidth * iCurrentPixely + i;
                //取得当前指针处的像素值，注意要转换为unsigned char型
                pixel = (unsigned char)*lpSrc;

                //有未填充的像素，将新的种子压入堆栈
                if (pixel == 255)
                {
                    StackPoint++;
                    Seeds[StackPoint].Height = iCurrentPixely;
                    Seeds[StackPoint].Width = i;
                    break;
                }
            }
        }

        // 返回
        return TRUE;
}
```

在 ch1_1view 中添加下面的事件处理代码：

```
void CCh1_1View::OnEdgeFill2()
{
    //种子填充运算

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR      lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的膨胀，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
```

```
        MessageBox("目前只支持256色位图的运算！","系统提示", MB_ICONINFORMATION |
MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用Fill2DIB()函数对DIB进行种子填充
    if (Fill2DIB(lpDIBBits, WIDTHBYTES(::DIBWidth(lpDIB) * 8), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！","系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();
}
```

图8－21和图8－22给出了程序运行的一个示例。

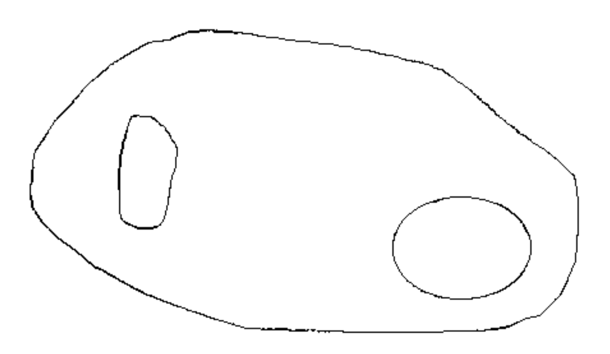


图 8－21 原始图像

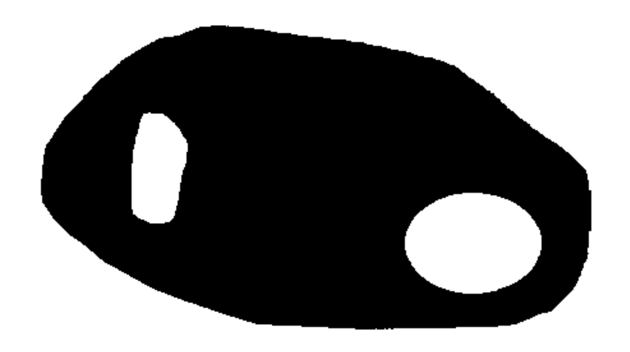


图 8－22 扫描线种子填充后的图像

第九章 图像分析

9.1 图像分割

9.1.1 基于幅度的图像分割

图像分割的目的是把图像空间分割成一些有意义的区域。例如：一幅航空照片，可以分割成工业区、住宅区及湖泊、森林等。这里“有意义”的内涵随着解决的问题的不同而不同。例如可以按幅度不同来分割各个区域；按边缘不同来划分各个区域；按形状来分割各个区域等等。

在多种分割方法中。我们首先介绍幅度分割。

幅度分割方法是把图像的灰度分成不同的等级，然后用设置灰度门限的方法确定有意义的区域或欲分割物体的边界。

假定一幅图像具有如图 9－1 所示的直方图。由直方图(a)可以知道图像 $f(x,y)$的大部分像素取值较低，其余像素较均匀地分布在其他灰度级上。由此可以推断这幅图像是由有灰度级的物体叠加在一个暗背景上形成的。可以设一个阈值 T，把直方图分成两个部分。如图 9－1(b)所示，T 的选择要本着如下的原则：B_1 应尽可能包含与背景相关连的灰度级，而 B_2 则应包含物体的所有灰度级。当扫描这幅图像时，从 B_1 到 B_2 之间的灰度变化就指示出有边界存在。为了找出水平方向上和垂直方向上的边界，应进行两次扫描。即首先确定一个门限 T，然后执行下列步骤：

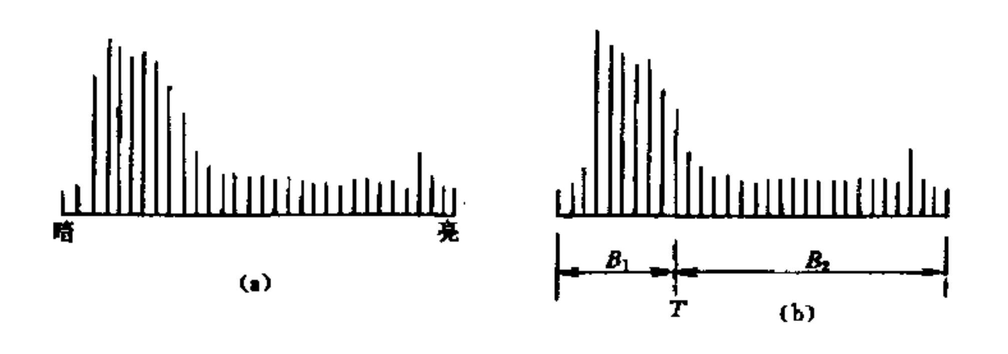


图 9－1 图像 f(x,y)的直方图

第一：对 $f(x,y)$的每行进行检测、产生的图像 $f_1(x,y)$的灰度将遵循如下规则：

如果 $f(x,y)$和 $f(x,y-1)$处在不同的灰度带上，则 $f_1(x,y) = L_E$;

否则，$f_1(x,y) = L_B$。

式中 L_E 是指定的边缘灰度级．L_B 是背景灰度级。

第二：对 $f(x,y)$ 的每一列进行检测，产生的图像 $f_2(x,y)$ 的灰度将遵循下列规则

如果 $f(x,y)$ 和 $f(x-1,y)$ 的灰度级处在不同的灰度带上，则 $f_2(x,y)=L_E$;

否则，$f_2x,y) = L_B$。

为了得到边缘图像，可采用下述关系：

如果 $f(x,y)$ 和 $f_2(x,y)$ 中的任何一个等于 L_E，则 $f(x,y)=L_E$;

否则，$f(x,y) = L_B$。

上述方法是以某像素到下一个像素间灰度的变化为基础的。这种方法也可以推广到多灰度级阈值方法中。由于确定了更多的灰度级阈值，可以提高边缘抽取技术的能力，其关键问题是如何选择阈值。

一种方法是把图像变成二值图像，如果图像 $f(x,y)$ 的灰度级范围是(Z_1，Z_k)，设 T 是 Z_1 和 Z_k 之间的一个数，那么 $f_t(x,y)$ 可由下式表示

$$f_t(x,y)=\begin{cases}1, f(x,y)\geq T\\ 0, f(x,y)<T\end{cases}$$

另一种方法是把规定的灰度级范围变换为 1，而范围以外的灰度变换为 0。例如

$$f_u(x,y)=\begin{cases}1, f(x,y)\leq u\\ 0, f(x,y)>u\end{cases}$$

$$f_{u,v}(x,y)=\begin{cases}0, f(x,y)<u\\ 1, u\leq f(x,y)\leq v\\ 0, f(x,y)>v\end{cases}$$

另外，还有一种半阈值法，是将灰度级低于某一阈值的像素灰度变换为零，而其余的灰度级不变。总之设置灰度级阈值的方法不仅可以提取物体的、也可以提取目标的轮廓。

上述方法都是以图像直方图为基础去设置阈值的。显然，从直方图上妥善地选择 T 值，对正确划分出感兴趣区域和背景是非常重要的。

如果已知分割正确的图像的一些特征，那么阈值确定只要试验不同的值，看是否满足特征即可。例如打印的纸张，如果已知打印的字符占一张纸上的面积的百分比，就可以找合适的阈值，使该条件得到满足。这就是最早使用的 P 片法，这种方法适用于事先知道目标面积比的情况。

如果前景物体的内部具有均匀一致的灰度值，并分布在另一个灰度值的均匀背景上，那么图像的灰度直方图将有明显的双峰。这种情况下可选择两峰之间的谷点作为门限值。这就是常说的峰谷法。该方法的优点是比较简单，但不适用于两峰值相差比较大、有宽且平的谷底的图像。

在许多情况下，噪声的干扰使谷的位置难以判定或者结果不稳定，这个问题在一定程度上需要对直方图进行平滑或曲线拟合来克服。图 9－2 所示为具有明显双峰的灰度直方图，可以用峰谷法对图像进行阈值分割。

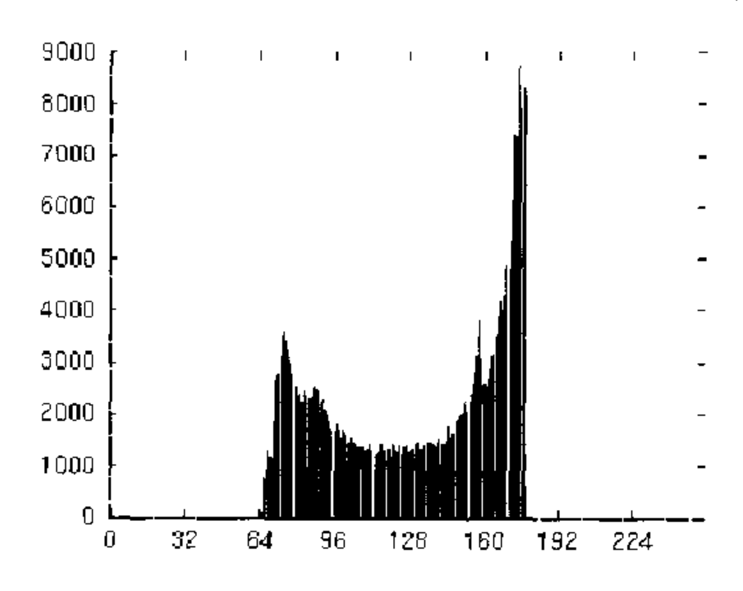


图 9－2 具有明显双峰的灰度直方图

下面介绍最小错误分割法。这种阈值分割方法的基本思想是找到一个门限阈值，使按这个阈值划分目标和背景的错误分割概率为最小。

设图像中感兴趣的目标的像素点灰度作正态分布，密度为 $P_1(c)$，均值和方差为 μ_1 和 σ_1^2，设背景点的灰度也作正态分布，密度为 $P_2(x)$，均值和方差为 μ_2 和 σ_2^2。换言之，整个密度函数可看作是两个单峰密度函数的混合。

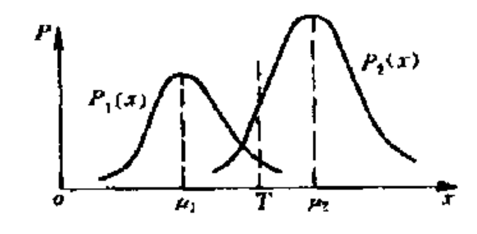


图 9－3 目标点和背景点的灰度分布

设目标的像点数占图像总点数的百分比为 Q，背景点占(1-Q)，则混合概率密度为：

$$P(X)=QP_1(x)+(1-Q)p_2(x)$$

$$=\frac{Q}{\sqrt{2\pi}\sigma_1}\exp\left[-\frac{(x-\mu_1)^2}{2\sigma_1^2}\right]+\frac{1-Q}{\sqrt{2\pi}\sigma_2}\exp\left[-\frac{(x-\mu_2)^2}{2\sigma_2^2}\right]$$

当选定门限为 T 时，目标点错划为背景点的概率为：

$$E_1(T)=\int_T^{\infty}P_1(x)\mathrm{d}X$$

把背景点错划为目标点的概率为：

$$E_2(T)=\int_{-\infty}^{T}P_2(x)dX$$

则总的的错误概率为：

$$E(T)=QE_1(T)+(1-Q)E_2(T)$$

令 $\dfrac{\partial E(T)}{\partial T}=0$

则有 $-QP_1(T)+(1-Q)P_2(T)=0$

由此得到 $\ln\dfrac{Q\sigma_2}{(1-Q)\sigma_1}-\dfrac{(T-\mu_1)^2}{2\sigma^2{}_1}=\dfrac{-(T-\mu_2)^2}{2\sigma^2{}_2}$

当 $\sigma_1^2=\sigma_2^2=\sigma^2$ 时

$$T=\frac{\mu_1+\mu_2}{2}+\frac{\sigma^2}{\mu_2-\mu_1}\ln\frac{Q}{1-Q}$$

若先验概率为已知，例如 $Q=1/2$，则有：

$$T=\frac{\mu_1+\mu_2}{2}$$

这表示正态分布时，最佳阈值可按上式求得。

对于复杂的图像，在许多情况下用单一的阈值不能得到良好的分割结果。在此种情况下，如果已知在图像上的位置函数描述不均匀照射可以设法用灰度级技术进行校正，然后采用单一阈值来分割；另外的方法是把图像分成小块，对每一块设置阈值。但是，如果某块图像只含物体或背景，那么这块图像就找不到阈值。这时，可以由附近的像象块求得的局部阈值用内插法给此像块指定一个阈值。

在确定阈值时，如果阈值定得过高，偶然出现的物体点就会被认作背景；如果阈值定得过低，则会发生相反的情况，克服的方法是使用两个阈值，例如 $t_1<t_2$，把灰度值超过 t_2 的像素分类为核心物体点，而灰度值超过 t_1 的像素仅当它们紧靠核心物体点时才算作物体点。t_2 的选择要使每个物体有一些像素灰度级高于 t_2，而背景不含有这样的像素。同时，应选择 t_1 使每个物体像素点具有高于 t_1 的灰度级。如果只使用 t_2 则物体总是分割得不完整；如果只使用 t_1 则会有许多背景像素被错分为物体像素；如果同时使用 t_1 和 t_2 就能把背景和物体很好地分割开来。当然，如果物体与背景的对比是鲜明的，就不必使用这种方法。

此外，如果存在一个阈值 t_2，使得每个物体的像素灰度级高于 t_2，而背景不包含这种像素，可对图像设置阈值 t_2，然后检查高于阈值像素的邻域，目的是寻找一个局部阈值。以便在每个类似邻域中把物体和背景分开。如果这些物体相当小，并且不太靠近时，这种方法比较适用。使用的邻域应足够大，以保证它们既包含物体像素，也包含背景像素，这样就可以使邻域的直方图是双峰的。

有时需要寻找一幅图像的局部最大点，即提取某种局部性质值比附近像素高的像素。一般来讲，也要求这些点具有一个高于低阈值 t_1 的值，一旦超过 t_1，不管它的绝对值大小如何，

一切相对的最大值都被采纳。因此，可以寻找局部最大值作为局部设置阈值的极端情况。在对图像进行匹配运算或检测界线时可采用这种方法。

由于图像的特征、图像的边缘和区域具有重要的意义，因此对边缘的检测和区域分割图像的分析和识别是至关重要的。进行边缘检测的最基本的方法是图像的微分(即差分)，梯度和拉普拉斯算子等方法，这些内容已经在第八章中叙述过，这里就不再重复了。

9.1.2 图像的区域分割

对于特征不连续的边缘检测，把图像分割成特征相同的互相不重叠区域的处理方法叫做区域分割。虽然目前已有许多方法，但还都不是特别具有决定性，因此有必要根据对象和目的不同而分别使用各种方法。前面所讲述的阈值处理，可以说是区域分割最简单的方法。下面介绍其他几种方法。

作为区域分割的方法，最基本的是区域扩张法。这种方法把图像分割成特征相同的小区域(最小的单位是像素)，研究与其相邻的各个小区域之间的特征。把具有类似特征的小区域依次合并起来。例如，为了从像素开始进行区域扩张，可操作如下：

1. 对图像进行光栅扫描，求出不属于任何区域的像素。
2. 把这个像素的灰度与其周围的(4 邻域或 8 邻域)不属于任何一个区域的像素灰度相比较，如果其差值在某一阈值以下，就把它作为同一个区域加以合并。
3. 对于那些新合并的像素，反复进行 2 的操作。
4. 反复进行 2、3 的操作，直至区域不能再扩张为止。
5. 返回到 1，寻找能成为新区域出发点的像素。

但用这样的方法，如图 9－4(a)那样的区域间边缘灰度将变化成为很平缓的场合，或者如图 9－4(b)那样对比度弱的边缘相交为一点时，两个区域会合并起来。为了消除这一缺点，在 2 的操作中，不是比较区域外围像素的灰度与其周围像素的灰度，而是比较已经存在的区域的平均灰度与该区域邻接像素的灰度值。

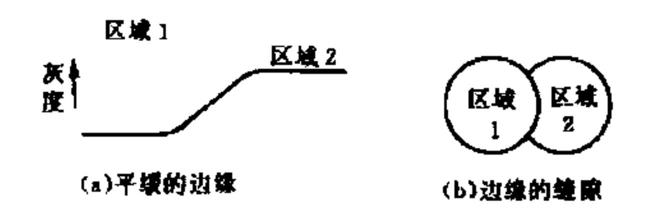


图 9－4 边缘对区域扩张的影响

但是，这样一来就会产生问题，无论从哪个像素起进行区域扩张，其最后的区域分割结果都将发生变化。

以下是不依赖于区域扩张起始点的方法。

1. 设灰度差的阈值＝0，按上述 1～5 操作进行区域扩张(使具有同一灰度的像素合并)。
2. 求出所有邻接区域的平均灰度差，合并具有最小灰度差的邻域区域组。
3. 通过反复进行 2 的操作，依次把区域合并。

用这种方法，如果在不适当的阶段停止区域合并，整幅画面最终就会成为一个区域。

以上的方法是把灰度差作为区域合并的判定标准的。此外，还有根据小区域内的灰度分布的相似性进行区域合并的方法，称为统计假说检测法。

1. 把图像分成相互稀疏的，大小为 $n \times n$ 的小矩形区域。
2. 比较邻接区域的灰度直方图，如果灰度分布情况都是相似的，就合并成一个区域。
3. 反复进行 2 的操作，直至区域合并完了为止。

检测灰度分布情况的相似性可采用下面的方法。设 $h_1(x)$，$h_2(x)$为相邻的两个区域的灰度直方图，从这个直方图求出累积灰度直方图 $H_1(x)$和 $H_2(x)$，根据以下两个准则：

(1)Kolmogorov-Smirnov 检测

$$\max_x \left| H_1(x) - H_2(x) \right|$$

(2)Smoothed-Difference 检测

$$\sum_x \left| H_1(x) - H_2(x) \right|$$

求出两者之差. 如果这个差在某一阈值以下，就把两个区域合并。这里灰度直方图 $h(x)$ 的累积灰度直方图 $H(x)$被定义为

$$H(x) = \int_0^x h(x)\mathrm{d}x$$

在数字化图像中，$H(x) = \sum_{i=o}^{x} h(i)$

根据上述的灰度分布相似性的区域扩张法，不仅能为分割灰度相同的区域使用，而且也能为分割具有纹理性的某个区域使用。但采用这种方法，把最初的 $n \times n$ 矩形区域作为单位，会出现下述情况：如果把 n 定大了，则区域的形状就变得不自然，小的对象物就会漏过；若把 n 定小了，可靠性就会减弱。所以 n 应常设在 5～10 的范围内。

以上所有的方法都是采用了仅仅与灰度有关的值作为区域合并的标准。另外，还有根据区域的形状作为判断标准的区域合并法。使用这种方法，首先把图像分割成灰度固定的区域，然后根据如下的评价函数进行区域合并。

1. 把任意的邻接区域 R_1、R_2 的周长设为 P_1、P_2，把在两个区域共同边界线两侧的灰度差在某一阈值 a 值以下的那部分长度设为 W。如果

$$W / \min\{P_1, P_2\} > \theta_1 \qquad \theta_1\text{：阈值}$$

则合并 R_1、R_2。

2. 在把 R_1，R_2 的共同边界的长度设为 B 的时候，如果

$$W / B > \theta_2 \qquad \theta_2\text{：阈值}$$

则合并 R_1、R_2。

1 的标准是为了合并得到一致的理想形状的标准，R_1、R_2 的共同边界在凸凹的情况下也易于被合并。而 2 则是合并共同边界中对比度低的部分比较多的区域的标准。

区域扩张法是基于重视图像空间的连通性而进行的区域分割方法，与此相反，还有根据像素的相似性在特征空间利用群聚进行的区域分割法。

这种方法把图 9－5 所示的像素或小区域所具有的特征映射到特征空间中，根据在待征空间的群聚，求出具有相似特性的像素或小区域。以后，各像素表示它所属群的标号。为了在图像空间最后求得各区域，有必要对编有标号的图像进行连通成份的编号码操作。图 9－5 带有标号 1 的像素，被区分为两个连通区域。这种方法是把像素或小区域作为 1 个图像的模式识别理论的应用。

对于图像内存在灰度或大小都不同的众多对象物的情况，简单的二值化方法起不到应有的作用。把简单的阈值处理加以扩充，可作为对复杂的图像进行区域分割的方法。这就是递归的阈值处理。这个方法是以彩色图像为对象开发的，需进行如下处理。

1. 从彩色图像求出对应于红、绿、蓝、亮度、色调、彩度等一共为 9 种特性的直方图。
2. 从各个直方图求出峰，并选择最突出的峰。取出被选为峰的像素，并从这些像素里求出连通区域。
3. 对于用 2 求得的连通区域(一般为多数个)以及其他剩余的连通区域，递归地反复进行分割直至对于所有特征的直方图完全成为单峰性为止。这里所谓的递归就是用 1、2 的操作而得到的某一区域，再进行 1、2 的处理，并分割成若干个区，再进一步对各个被细分的连通区域进行 1、2 的处理，如此逐次地把区域细分反复进行下去。

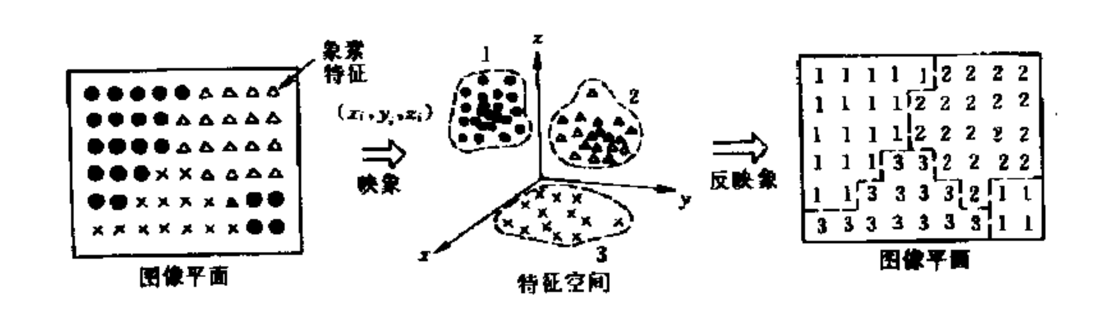


图 9－5 在特征空间中考虑群聚的区域分割

9.1.3 Visual C++编程实现

下面我们介绍一种迭代求图像最佳分割阈值的算法，并编程使之实现。

这一算法的步骤如下：

1. 求出图像中的最小和最大灰度值 Z_l 和 Z_k，令阈值初值

$$T^0 = \frac{Z_l + Z_k}{2}$$

2. 根据阈值 T^K 将图像分割成目标和背景两部分，求出两部分的平均灰度值 Z_0 和 Z_B：

$$Z_O = \frac{\sum_{z(i,j)<T^k} z(i,j) \times N(i,j)}{\sum_{z(i,j)<T^k} N(i,j)}$$

$$Z_B = \frac{\sum_{z(i,j)>T^k} z(i,j) \times N(i,j)}{\sum_{z(i,j)>T^k} N(i,j)}$$

式中$Z(i,j)$是图像上(i,j)点的灰度值，$N(i,j)$是 (i,j)点的权重系数，一般$N(i,j)=1.0$。

3. 求出新的阈值：

$$T^{K+1} = \frac{Z_O + Z_B}{2}$$

4. 如果$T^K=T^{K+1}$，则结束，否则$K \leftarrow K+1$，转步2。

根据上面所述的算法，我们编写了对图像进行阈值分割的函数 ThresholdDIB()，位于文件 detect.cpp 中：

```
// ************************************************************************
//  文件名：detect.cpp
//
//  图像分析与检测API函数库：
//
//  ThresholdDIB()  - 图像阈值分割运算
//  AddMinusDIB()   - 图像加减运算
//  HprojectDIB()   - 图像水平投影
//  VprojectDIB()   - 图像垂直投影
//  TemplateDIB()   - 图像模板匹配运算
//
// ************************************************************************

#include "stdafx.h"
#include "detect.h"
#include "DIBAPI.h"
#include <math.h>
#include <direct.h>

/*************************************************************************
 *
 * 函数名称：
 *   ThresholdDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
```

```
 * 返回值:
 *   BOOL                 - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行阈值分割运算。
 *
 ************************************************************************/

BOOL WINAPI ThresholdDIB(LPSTR lpDIBBits,LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL  hNewDIBBits;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel;

    //直方图数组
    long lHistogram[256];

    //阈值，最大灰度值与最小灰度值，两个区域的平均灰度值
    unsigned char
iThreshold,iNewThreshold,iMaxGrayValue,iMinGrayValue,iMean1GrayValue,iMean2GrayValue;

    //用于计算区域灰度平均值的中间变量
    long lP1,lP2,lS1,lS2;

    //迭代次数
    int iIterationTimes;

    // 图像每行的字节数
    LONG lLineBytes;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
```

```
	}

	// 锁定内存
	lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

	// 初始化新分配的内存，设定初始值为255
	lpDst = (char *)lpNewDIBBits;
	memset(lpDst, (BYTE)255, lWidth * lHeight);

	// 计算图像每行的字节数
	lLineBytes = WIDTHBYTES(lWidth * 8);

	for (i = 0; i < 256;i++)
	{
		lHistogram[i]=0;
	}

	//获得直方图
	iMaxGrayValue = 0;
	iMinGrayValue = 255;
	for (i = 0;i < lWidth ;i++)
	{
		for(j = 0;j < lHeight ;j++)
		{
			// 指向原图像倒数第j行，第i个像素的指针
			lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

			pixel = (unsigned char)*lpSrc;

			lHistogram[pixel]++;
			//修改最大，最小灰度值
			if(iMinGrayValue > pixel)
			{
				iMinGrayValue = pixel;
			}
			if(iMaxGrayValue < pixel)
			{
				iMaxGrayValue = pixel;
			}
		}
	}

	//迭代求最佳阈值
	iNewThreshold = (iMinGrayValue + iMaxGrayValue)/2;
	iThreshold = 0;

	for(iIterationTimes = 0; iThreshold != iNewThreshold && iIterationTimes < 
100;iIterationTimes ++)
	{
		iThreshold = iNewThreshold;
		lP1 =0;
```

```
        lP2 = 0;
        lS1 = 0;
        lS2 = 0;
        //求两个区域的灰度平均值
        for (i = iMinGrayValue;i < iThreshold;i++)
        {
            lP1 += lHistogram[i]*i;
            lS1 += lHistogram[i];
        }
        iMean1GrayValue = (unsigned char)(lP1 / lS1);
        for (i = iThreshold+1;i < iMaxGrayValue;i++)
        {
            lP2 += lHistogram[i]*i;
            lS2 += lHistogram[i];
        }
        iMean2GrayValue = (unsigned char)(lP2 / lS2);
        iNewThreshold =  (iMean1GrayValue + iMean2GrayValue)/2;
    }

    //根据阈值将图像二值化
    for (i = 0;i < lWidth ;i++)
    {
        for(j = 0;j < lHeight ;j++)
        {
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            // 指向目标图像倒数第j行，第i个像素的指针
            lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;

            pixel = (unsigned char)*lpSrc;

            if(pixel <= iThreshold)
            {
                *lpDst = (unsigned char)0;
            }
            else
            {
                *lpDst = (unsigned char)255;
            }
        }
    }

    // 复制图像
    memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
```

```
    return TRUE;
}
```

下面在程序中添加一个图像分析的菜单。如图 9—6 所示：

图 9—6 图像分析菜单

在 ch1_1view.cpp 中添加菜单事件的处理代码：

```
void CCh1_1View::OnDetectThreshold()
{
    //阈值分割

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的阈值分割，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }
    // 更改光标形状
    BeginWaitCursor();

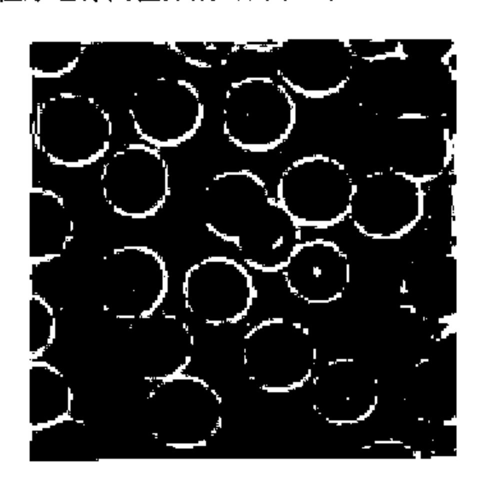
    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用ThresholdDIB()函数对DIB进行阈值分割
    if (ThresholdDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {
```

```
        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }
    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
    // 恢复光标
    EndWaitCursor();
}
```

下面来看一下用这个程序进行阈值分割的例子。如图 9—7 及图 9—8 所示：



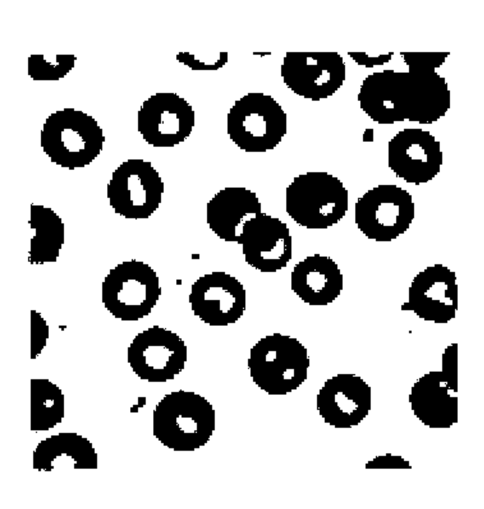


图 9—8 阈值分割的结果图像

9.2 投影法与差影法

9.2.1 投影法

我们用一个具体的例子来介绍投影法。下面的这幅照片是著名的华盛顿纪念碑，利用投影法，可以从图 9－9 中自动检测到水平方向上纪念碑的位置。



图 9－9 华盛顿纪念碑

仔细观察这幅图像，可以发现，纪念碑上像素的灰度都差不多，而且和其他区域的灰度值不同。如果我们选取合适的阈值，做削波处理（这里选 175 到 220），将该图二值化，就可以把纪念碑突出显示出来。如图 9－10 所示：



图 9－10 削波处理，将图 9－9 二值化

由于纪念碑所在的那几列的白色点比起其他列多得多，如果把该图在竖直方向做投影，则如图 9－11 所示：其中黑色线条的高度代表了该列上白色点的个数。图中间的高峰部分就是要找的水平方向上纪念碑所在的位置，这就是投影法。

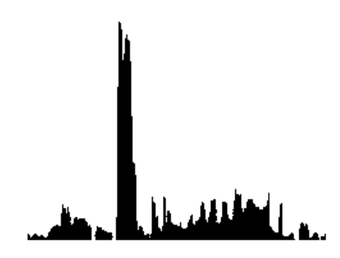


图 9-11 作竖直方向投影

可以看出投影法是一种很自然的方法，有点象灰度直方图。为了得到更好的效果，投影法经常和阈值化一起使用。由于噪声点对投影有一定的影响，所以处理前最好先做一次平滑，去除噪声。

9.2.2 图像的代数运算与差影法

代数运算是指对两幅输入图像进行点对点的加、减、乘、除四则运算而得到输出图像的运算。

四种图像处理代数运算的数学表达式如下：

$$C(x, y) = A(x, y) + B(x, y)$$

$$C(x, y) = A(x, y) - B(x, y)$$

$$C(x, y) = A(x, y) \times B(x, y)$$

$$C(x, y) = A(x, y) \div B(x, y)$$

其中 $A(x,y)$ 和 $B(x,y)$ 为输入图像，而 $C(x,y)$ 为输出图像。还可通过适当的组合，形成涉及几幅图像的复合代数运算方程。

图像相加的一个重要应用是对同一场景的多幅图像求平均值。这点经常被用来有效地降低加性随机噪声的影响。图像相加也可以将一幅图像的内容叠加到另一幅图像上去，以达到二次曝光的效果。

图像相减可用于去除一幅图像中不需要的加性图案，加性图案可能是缓慢变化的背景阴影、周期性的噪声或是在图像上每一像素处均已知的附加污染等。减法也可用于检测同一场景的两幅图像之间的变化。例如：通过对一场景的序列图像的减运算可用于检测运动等。

计算用于确定物体边界位置的梯度也要用到图像减运算。

在数字图像处理中，乘和除用得较少一些，但它们也有着很重要的用途。量化器对一幅图像各点的敏感程度有可能造成变化，乘和除运算有可能用于纠正这种影响。除运算可产生对颜色和多光谱图像分析十分重要的比率图像。用一幅掩膜图像乘某一图像可遮住该图像中

的某些部分时，使其仅留下感兴趣的物体。

下面我们将讨论图像加运算及减运算后的输出直方图，这将使我们更深入地理解各种运算以及为了保证输出灰度不越界而进行的灰度伸缩的必要性。

在图像的加运算中，假定输入图像 $A(x,y)$和 $B(x,y)$的灰度直方图分别为 $H_A(D)$和 $H_B(D)$。我们希望得到输出图像的直方图 $H_C(D)$。如果输入图像是相同的，或其中之一为常数，代数运算就归结为点运算。在本节，我们将讨论的是两幅图像互不相关的情形。

如果两幅输入图像的联合二维直方图是各自的直方图之积，即：

$$H_{AB}(D_A,D_B)=H_A(D_A)H_B(D_B)$$

则称该两幅图像是不相关的。实际上，这意味着两图像之间没有任何关系。

如果输入图像是完全相同的，则这一等式将不成立。但是，如果至少有一幅图像是随机的并且在统计的意义上独立于另一幅图像，则该等式成立。

我们可通过对其中一独立的自变量进行积分，将二维直方图降为一维的边际直方图，也就是

$$H(D_A)=\int_{-\infty}^{+\infty}H_{AB}(D_A,D_B)\mathrm{d}D_B$$

将上式代入，我们可得到一维直方图：

$$H(D)=\int_{-\infty}^{+\infty}H_A(D_A)H_B(D_B)\mathrm{d}D_B$$

根据图像加法运算的定义可知，在图像上每一个点有：

$$D_A=D_C-D_B$$

代入到上式中，可得：

$$H(D)=\int_{-\infty}^{+\infty}H_A(D_C-D_B)H_B(D_B)\mathrm{d}D_B$$

该一维直方图是输出灰度级的函数，因此它是输出直方图。现在，我们可将两幅不相关的图像进行加运算后所得到的输出直方图表示为：

$$H_C(D_C)=H_A(D_A)*H_B(D_B)$$

符号*代表卷积运算。

下面我们看一个简单的例子。假定我们希望对两个完全相同的、形式为 e^{-x^2} 的高斯函数进行卷积操作，则：

$$\mathrm{e}^{-x^2}*\mathrm{e}^{-x^2}=\int_{-\infty}^{+\infty}\mathrm{e}^{-y^2}\mathrm{e}^{-(x-y)^2}\mathrm{d}y$$

展开指数项进行相关项合并，可得：

$$e^{-x^2} * e^{-x^2} = \int_{-\infty}^{+\infty} e^{-(x^2-2xy-2y^2)} \mathrm{d}y$$

我们可以在其中插入乘积为 1 的项，得到：

$$e^{-x^2} * e^{-x^2} = \int_{-\infty}^{+\infty} e^{-(x^2-2xy-2y^2)} e^{+x^2/2} e^{-x^2/2} \mathrm{d}y$$

经整理后可得：

$$e^{-x^2} * e^{-x^2} = \int_{-\infty}^{+\infty} e^{-2(x^2/4-xy+y^2)} e^{-x^2/2} \mathrm{d}y$$

对指数进行因式分解，可得：

$$e^{-x^2} * e^{-x^2} = \int_{-\infty}^{+\infty} e^{-2(y-x/2)^2} e^{-x^2/2} \mathrm{d}y$$

整理后可得：

$$e^{-x^2} * e^{-x^2} = e^{-x^2/2} \int_{-\infty}^{+\infty} e^{\frac{-(y-x/2)^2}{2\times\frac{1}{4}}} \mathrm{d}y$$

我们可利用如下高斯函数的性质，即：

$$\int_{-\infty}^{+\infty} e^{\frac{-(x-\mu)^2}{2\sigma^2}} \mathrm{d}x = \sqrt{2\pi\sigma^2}$$

显然，上式将变成：

$$e^{-x^2} * e^{-x^2} = \sqrt{2\pi\frac{1}{4}} e^{-x^2/2}$$

类似地，可导出在更普遍的情况下，

$$A_1 e^{\frac{-(x-\mu_1)^2}{2\sigma_1^2}} * A_2 e^{\frac{-(x-\mu_2)^2}{2\sigma_2^2}} = A_1 A_2 \sqrt{2\pi\sigma_1\sigma_2} e^{\frac{-(x-\mu_3)^2}{2\sigma_3^2}}$$

其中

$$\mu_3 = \mu_1 + \mu_2$$

$$\sigma_3^2 = \sigma_1^2 + \sigma_2^2$$

这意味着对两个高斯函数进行卷积运算，产生了第三个高斯函数，后者的均值发生了移动，并且方差被扩大了。

一般来说，卷积会使函数变“模糊”。因为将两幅无关图像相加意味着将它们的直方图进行卷积，经过图像相加运算所得到的图像将比两个原图像占有更大的灰度级范围。

对于两幅图像之间的相减运算，我们可以重新定义一幅图像，该图像为原图像求反所得。这样两幅图像求差的运算可以转化为求和运算，并且可以利用上面所得的结果。因此，对不相关图像的加和减运算在操作上是一样的。

然而，对于减法运算，有一种情况需要更进一步的考虑，即，两幅几乎完全相同但稍微有些不对准的图像的减法运算。当将一个场景中系列图像相减以用来检测运动或其他变化时，准确对是难以得到保证的。在这种情况下就需要更进一步的考虑。

假定差图像由下式给定：

$$C(x, y) = A(x, y) - A(x + \Delta x, y)$$

如果Δx很小，那么上式可以近似为

$$C(x, y) \approx \frac{\partial}{\partial x} A(x, y) \Delta x$$

注意到$\frac{\partial A}{\partial x}$本身也是一幅图像，我们将其直方图以$H'_A(D)$表示。因此，由上式表示的位移差图像的直方图为

$$H_C(D) \approx \frac{1}{\Delta x} H'_A(D / \Delta x)$$

因此，减去稍微有些不对准的一幅图像的复制品可得到偏导数图像。偏导数的方向为图像位移的方向。

所谓差影法实际上就是图像的相减运算（又称减影技术），是指把同一景物在不同时间拍摄的图像或同一景物在不同波段的图像相减。差值图像提供了图像间的差异信息，能用以指导动态监测、运动目标检测和跟踪、图像背景消除及目标识别等工作。

在利用遥感图像进行动态监测时，用差值图像可发现森林火灾、洪水泛滥及监测灾情变化，估计损失等；也能用以监测河口、海岸的泥沙淤积及监视江河、湖泊、海岸等的污染。利用差值图像还能发现图像上的云和阴影，鉴别出耕作地及不同的作物覆盖情况；利用同一地面上的物体在各波段的亮度差异还可以识别地面上的物体。

利用减影技术消除图像背景也有很明显的效果。典型的有医学上的应用，如在血管造影技术中肾动脉造影术对诊断肾脏疾病就有独特效果。为了减少误诊，人们希望提供反映游离血管的清晰图像。通常的肾动脉造影在造影剂注入后，虽然能够看出肾动脉血管的形状及分布，但由于肾脏周围血管受到脊椎及其他组织影像的重叠，难以得到理想的游离血管图像。对此人们摄制出肾动脉造影前后两幅图像，相减后就能把脊椎及其他组织的影像去掉，而仅保留血管图像。若再作对比度增强及彩色增强，就能得到清晰的游离血管图像。类似的技术也可用于诊断印刷线路板及集成电路掩模的缺陷当中。

图像在作差影法运算时必须使两相减图像的对应像点位于空间同一目标点上；若不是，必须先作几何校正与配准。

9.2.3 Visual C++编程实现

投影法的程序包括两个：水平方向投影子程序 HprojectDIB()和垂直方向投影子程序 VprojectDIB()。两个子程序均包括在 detect.cpp 中。

```
/*************************************************************************
 *
 * 函数名称：
 *   HprojectDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对两幅图像进行水平投影运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI HprojectDIB(LPSTR lpDIBBits,LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL  hNewDIBBits;

    //循环变量
    long i;
    long j;

    //图像中每行内的黑点个数
    long lBlackNumber;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);
```

```
if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化新分配的内存，设定初始值为255
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lWidth * lHeight);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

for (j = 0;j < lHeight ;j++)
{
    lBlackNumber = 0;
    for(i = 0;i < lWidth ;i++)
    {
        // 指向原图像倒数第j行，第i个像素的指针
        lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

        pixel = (unsigned char)*lpSrc;

        if (pixel != 255 && pixel != 0)
        {
            return false;
        }
        if(pixel == 0)
        {
            lBlackNumber++;
        }
    }
    for(i = 0;i < lBlackNumber ;i++)
    {
        // 指向目标图像倒数第j行，第i个像素的指针
        lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;

        *lpDst = (unsigned char)0;
    }
}


// 复制投影图像
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

// 释放内存
LocalUnlock(hNewDIBBits);
```

```
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   VprojectDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对两幅图像进行垂直投影运算。
 *
 * 要求目标图像为只有0和255两个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI VprojectDIB(LPSTR lpDIBBits,LONG lWidth, LONG lHeight)
{

    // 指向原图像的指针
    LPSTR   lpSrc;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL  hNewDIBBits;

    //循环变量
    long i;
    long j;

    //图像中每行内的黑点个数
    long lBlackNumber;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;
```

```
// 暂时分配内存，以保存新图像
hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

if (hNewDIBBits == NULL)
{
    // 分配内存失败
    return FALSE;
}

// 锁定内存
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化新分配的内存，设定初始值为255
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lWidth * lHeight);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

for (i = 0;i < lWidth ;i++)
{
    lBlackNumber = 0;
    for(j = 0;j < lHeight ;j++)
    {
        // 指向原图像倒数第j行，第i个像素的指针
        lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

        pixel = (unsigned char)*lpSrc;

        if (pixel != 255 && pixel != 0)
        {
            return false;
        }
        if(pixel == 0)
        {
            lBlackNumber++;
        }
    }
    for(j = 0;j < lBlackNumber ;j++)
    {
        // 指向目标图像倒数第j行，第i个像素的指针
        lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;

        *lpDst = (unsigned char)0;
    }
}


// 复制投影图像
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);
```

```
    // 释放内存
    LocalUnlock(hNewDIBBits);
    LocalFree(hNewDIBBits);

    // 返回
    return TRUE;
}
```

对应的菜单单击事件处理函数如下：

```
void CCh1_1View::OnDetectHprojection()
{
    //水平投影

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的投影，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！","系统提示", MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用HprojectDIB()函数对DIB进行水平投影
    if (HprojectDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
```

```
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
void CCh1_1View::OnDetectVprojection()
{
    //垂直投影

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的投影，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用VprojectDIB()函数对DIB进行垂直投影
```

```
    if (VprojectDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

下面是水平投影和垂直投影的运行结果。如图9－12、图9－13和图9－14所示。

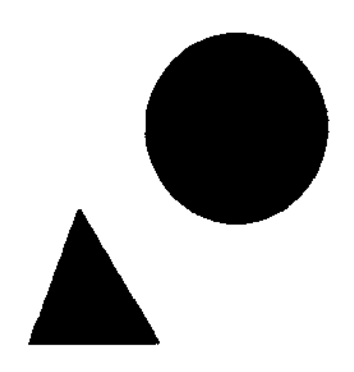


图 9—12 原始图像



图9－13 水平投影结果



图9－14 垂直投影结果

图像的加法和减法运算由函数AddMinusDIB()实现。该函数的原型为：

```
BOOL WINAPI AddMinusDIB (LPSTR lpDIBBits, LPSTR lpDIBBitsBK, LONG lWidth, LONG lHeight, bool
bAddMinus);
```

其中lpDIBBits是指向原始图像的指针，lpDIBBitsBK是指向背景图像的指针，bAddMinus是选择做加法还是做减法的逻辑变量。如果bAddMinus=true，那么做加法运算；否则做减法运算。

AddMinusDIB()函数如下：

```
/*************************************************************************
 *
 * 函数名称:
 *   AddMinusDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits     - 指向原DIB图像指针
 *   LPSTR lpDIBBitsBK   - 指向背景DIB图像指针
 *   LONG  lWidth        - 原图像宽度（像素数）
 *   LONG  lHeight       - 原图像高度（像素数）
 *   bool  bAddMinus     - 为true时执行加运算，否则执行减运算。
 *
```

```
 * 返回值:
 *   BOOL               - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对两幅图像进行加减运算。
 *
 * 要求目标图像为255个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI AddMinusDIB(LPSTR lpDIBBits, LPSTR lpDIBBitsBK, LONG lWidth, LONG lHeight ,bool
bAddMinus)
{

    // 指向原图像的指针
    LPSTR   lpSrc,lpSrcBK;

    // 指向缓存图像的指针
    LPSTR   lpDst;

    // 指向缓存DIB图像的指针
    LPSTR   lpNewDIBBits;
    HLOCAL  hNewDIBBits;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel,pixelBK;

    // 图像每行的字节数
    LONG lLineBytes;

    // 暂时分配内存，以保存新图像
    hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

    if (hNewDIBBits == NULL)
    {
        // 分配内存失败
        return FALSE;
    }

    // 锁定内存
    lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

    // 初始化新分配的内存，设定初始值为255
    lpDst = (char *)lpNewDIBBits;
    memset(lpDst, (BYTE)255, lWidth * lHeight);

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);
```

```
        for (j = 0;j < lHeight ;j++)
        {
            for(i = 0;i < lWidth ;i++)
            {
                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lLineBytes * j + i;
                lpSrcBK = (char *)lpDIBBitsBK + lLineBytes * j + i;

                // 指向目标图像倒数第j行，第i个像素的指针
                lpDst = (char *)lpNewDIBBits + lLineBytes * j + i;

                pixel = (unsigned char)*lpSrc;
                pixelBK = (unsigned char)*lpSrcBK;
                if(bAddMinus)
                    *lpDst = pixel - pixelBK > 255 ? 255 : pixel + pixelBK;
                else
                    *lpDst = pixel - pixelBK < 0 ? 0 : pixel - pixelBK;


            }
        }


        // 复制腐蚀后的图像
        memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

        // 释放内存
        LocalUnlock(hNewDIBBits);
        LocalFree(hNewDIBBits);

        // 返回
        return TRUE;
    }
```

下面为相应的菜单添加单击事件处理函数。当用户选择“差影法”时，首先要求用户选择一副背景图像，然后在事件处理函数中打开这幅背景图像，令 **lpDIBBitsBK** 指向背景图像，再调用 **AddMinusDIB()**函数做减法运算。

```
    void CCh1_1View::OnDetectMinus()
    {
        // 获取文档
        CCh1_1Doc* pDoc = GetDocument();

        // 指向DIB的指针
        LPSTR lpDIB,lpDIBBK;

        // 指向DIB像素指针
        LPSTR     lpDIBBits,lpDIBBitsBK;

        //图像的宽度与高度
        long lWidth,lHeight;
```

```
HDIB hDIBBK;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());


// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的水平镜像，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的平移！", "系统提示", MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}


// 更改光标形状
BeginWaitCursor();

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

lWidth = ::DIBWidth(lpDIB);
lHeight = ::DIBHeight(lpDIB);
CFileDialog dlg(TRUE,"bmp","*.bmp");
if(dlg.DoModal() == IDOK)
{

    CFile file;
    CFileException fe;

    CString strPathName;

    strPathName = dlg.GetPathName();

    // 打开文件
    VERIFY(file.Open(strPathName, CFile::modeRead | CFile::shareDenyWrite, &fe));

    // 尝试调用ReadDIBFile()读取图像
    TRY
    {
        hDIBBK = ::ReadDIBFile(file);
    }
    CATCH (CFileException, eLoad)
    {
        // 读取失败
```

```
            file.Abort();

            // 恢复光标形状
            EndWaitCursor();

            // 报告失败
            //ReportSaveLoadException(strPathName, eLoad,
//                FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

            // 设置DIB为空
            hDIBBK = NULL;

            // 返回
            return;
        }
        END_CATCH

        // 初始化DIB
        //InitDIBData();

        // 判断读取文件是否成功
        if (hDIBBK == NULL)
        {
            // 失败，可能非BMP格式
            CString strMsg;
            strMsg = "读取图像时出错！可能是不支持该类型的图像文件！";

            // 提示出错
            MessageBox(strMsg, NULL, MB_ICONINFORMATION | MB_OK);

            // 恢复光标形状
            EndWaitCursor();

            // 返回
            return;
        }
    }
    else
    {
        // 恢复光标形状
        EndWaitCursor();

        return;
    }
    // 锁定DIB
    lpDIBBK = (LPSTR) ::GlobalLock((HGLOBAL) hDIBBK);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的差影，其他的可以类推）
    if (::DIBNumColors(lpDIBBK) != 256)
    {
        // 提示用户
```

```
        MessageBox("目前只支持256色位图！", "系统提示" , MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDIBBK);

        // 返回
        return;
    }


    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBitsBK = ::FindDIBBits(lpDIBBK);

    // 调用AddMinusDIB()函数相减两幅DIB
    if (AddMinusDIB(lpDIBBits,lpDIBBitsBK, lWidth,lHeight,false))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
    ::GlobalUnlock((HGLOBAL) hDIBBK);

    // 恢复光标
    EndWaitCursor();
}
```

下面是运用差影法对一幅混叠图像进行处理的例子。如图9－15、图9－16和图9－17所示。



图 9-15 原始混叠图像



图 9-16 背景图像



图 9-17 去背景处理的结果图像

9.3 图像的匹配

9.3.1 模板匹配法

在机器识别事物的过程中，常需把不同传感器或同一传感器在不同时间、不同成像条件下对同一景物获取的两幅或多幅图像在空间上对准，或根据已知模式到另一幅图中寻找相应的模式，这就叫做匹配。在遥感图像处理中需把不同波段传感器对同一景物拍的多光谱图像按象点对应套准，然后根据像点的性质进行地物分类。如果利用在不同时间对同一地面拍摄得两幅照片，经套准后找出其中特征有了变化的像点，就可以用来分析图中哪些部分发生了变化；而利用放在一定间距处的两只传感器对同一物体摄得的两幅照片，找出对应点后可算出物体离开摄像机的距离，即深度信息。在其他方面如对序列图像匹配求光流场、描述三维动态景物、计算物体的空间结构和运动参量等匹配技术都起着至关重要的作用，自然地受到人们的重视，现已提出众多的匹配方法。

早期的图像匹配技术主要用于几何校正后的多波段遥感图像的套准，借助于求互相关函数的极值来实现。但在三维景物分析中，由于三维成像中有透视失真、运动遮挡及阴影混入和噪声干扰等不利因素，三维图像匹配至今仍是公认的技术难题，下面就一些代表性的方法作一些分析。

前面在对图像进行边缘锐化及检测的时候我们提到了模板匹配的方法，这里讨论图像的匹配，也可以用模板匹配法来研究在一幅图中是否存在某种已知模板图像。例如在图 9－18(a)中，需要寻找一下有无三角形（图 9－17(b)）的图像。若在被搜索图中有待寻的目标，且同模板有一样的尺寸和方向，它的基本原则就是通过相关函数的计算来找到它以及被搜索图的坐标位置。

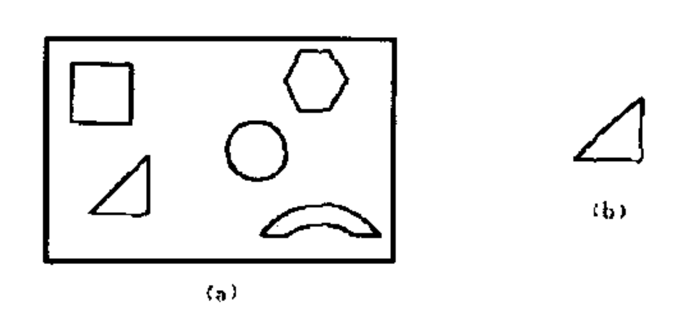


图 9－18 被搜索图(a) 与 模板(b)

设模板 T 叠放在搜索图 S 上平移，模板覆盖下的那块搜索图叫做子图 $S^{i,j}$，i，j 为这块子图的左上角象点在 S 图中的坐标，叫参考点，可从图 9-19 中看出 i 和 j 的取值范围为

$$1 < i, j < N - M + 1$$

现在可以比较 T 和 $S^{i,j}$ 的内容。若两者一致，则 T 和 $S^{i,j}$ 之差为零、所以可以用下列两种两种测度之一来衡量 T 和 $S^{i,j}$ 的相似程度：

$$D(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{M}[S^{i,j}(m,n)-T(m,n)]^2$$

或者

$$D(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{M}|S^{i,j}(m,n)-T(m,n)|$$

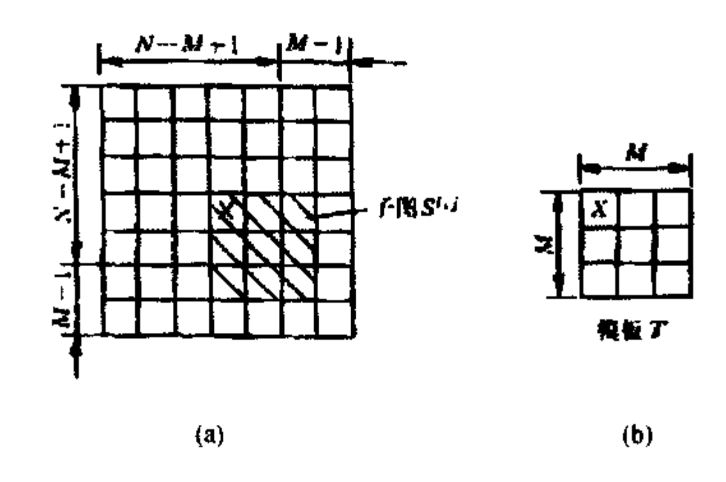


图 9－19 模板(b)及其搜索图(a)

如果展开前一个式子，则有

$$D(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{M}[S^{i,j}(m,n)]^2-2\sum_{m=1}^{M}\sum_{n=1}^{M}S^{i,j}(m,n)\times T(m,n)+\sum_{m=1}^{M}\sum_{n=1}^{M}[T(m,n)]^2$$

右边第三项表示模板的总能量，是一个常数与(i,j)无关，第一项是模板覆盖下那块图像子图的能量，它随(i,j)位置而缓慢改变，第二项是子图像和模板的互相关，随(i,j)而改变。T 和 $S^{i,j}$ 匹配时这一项的取值最大，因此我们可以用下列相关函数作相似性测度：

$$R(i,j)=\frac{\sum_{m=1}^{M}\sum_{n=1}^{M}S^{i,j}(m,n)\times T(m,n)}{\sum_{m=1}^{M}\sum_{n=1}^{M}[S^{i,j}(m,n)]^2}$$

或者归一化为

$$R(i,j)=\frac{\sum_{m=1}^{M}\sum_{n=1}^{M}S^{i,j}(m,n)\times T(m,n)}{\sqrt{(\sum_{m=1}^{M}\sum_{n=1}^{M}[S^{i,j}(m,n)]^2)}\sqrt{(\sum_{m=1}^{M}\sum_{n=1}^{M}[T(m,n)]^2)}}$$

根据施瓦兹不等式可以知道上式中 $0<R(i,j)<1$，并且仅在比值 $\dfrac{S^{i,j}(m,n)}{T(m,n)}$ 为常数时 $R(i,j$ 取极大值(等于 1)。上式可写成更简洁的内积形式，令 $S_1(i,j)$表示子图，t 表示模板。则有

$$R(i,j)=\frac{t^T S_1(i,j)}{\sqrt{(t^T t)}\sqrt{(S_1^{\ T}(i,j)S_1(i,j))}}$$

当矢量 t 和 S_1 之间的夹角为零时，即 $S_1(i,\ j)=kt$ 时(这里 k 为标量常数)，有 $R(i,\ j)=1$，否则有 $R(i,\ j)<1$。用相关法求匹配的计算量很大，因为模板要在$(N-M+1)$个参考位置上做相关计算。其中除一点以外都是在非匹配点上做无用功。因此，我们希望有一种快速计算方法。下面我们介绍一类叫序贯相似性检测的算法，简称 SSDA，其要点是：

1. 定义绝对误差值：

$$\varepsilon(i,j,m_k,n_k)=|S^{ij}(m_k,n_k)-\hat{S}(i,j)-T(m_k,n_k)+\hat{T}$$

式中，

$$\hat{S}(i,j)=\frac{1}{M^2}\sum_{m=1}^{M}\sum_{n=1}^{M}S^{i,j}(m,n)$$

$$\hat{T}(i,j)=\frac{1}{M^2}\sum_{m=1}^{M}\sum_{n=1}^{M}T(m,n)$$

2. 取一不变阈值 T_k。
3. 在子图 $S^{i,j}(m,n)$中随机选取像点。计算它同 T 中对应点的误差值 ε，然后把这差值同其他点对的差值累加起来，当累加 r 次误差超过 T_k，则停止累加，并记下次数 r，定义 SSDA 的检测曲面为

$$I(i,j)=\{r\mid \min_{1\le r\le m^2}[\sum_{k=1}^{r}\varepsilon(i,j,m_k,n_k)\ge T_k]\}$$

4. 把 I(i,j)值大的(i,j)点作为匹配点，因为这点上需要很多次累加才使总误差 Σε 超过 T_k，如图 9－18 所示。图中给出了在 A、B、C 三参考点上得到的误差累计增长曲线。A、B 反映模板 T 不在匹配点上，这时 Σε 增长很快，超出阈值。曲线 C 中 Σε 增长很慢，很可能是一套准的候选点。

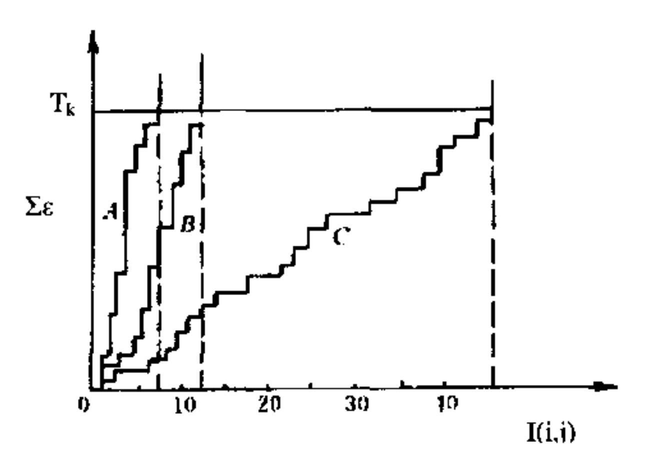


图 9－20 T_K＝常数时的累计误差增长曲线

SSDA 算法还可以进一步改进计算效率，办法是：

1. 对于(*N*-*M*+1)个参考点的选用顺序可以不逐点推进，即模扳不一定对每点都平移到。例如可用粗—细结合的均匀搜索，即先每隔 *m* 点搜索一下匹配好坏，然后在有极大匹配值周围的局部范围内对各参考点位置求匹配。这一策略能否不丢失真正匹配点，将取决于表面 *I*(*i*. *j*)的平滑性和单峰性。
2. 在某参考点(*i*, *j*)处，对模板覆盖下的 M^2 个点，可用与 *i*、*j* 无关的随机方式决定计算误差的先后顺序。也可以采用适应图像内容的方式，按模板中突出特征选取伪随机序列，决定计算误差的先后顺序，以便及早抛弃那些非匹配点。
3. 模板在(*i*, *j*)点得到的累计误差映射为上述曲面数值的方法，是否为最佳方法还有待探索。
4. 不选用固定阈值 T_k，而改用单调增长的阈值序列，使得非匹配点在更少的计算过程中就达到阈值而被丢弃，真匹配点则需更多次误差累计才达到阈值(见图 9—21)。

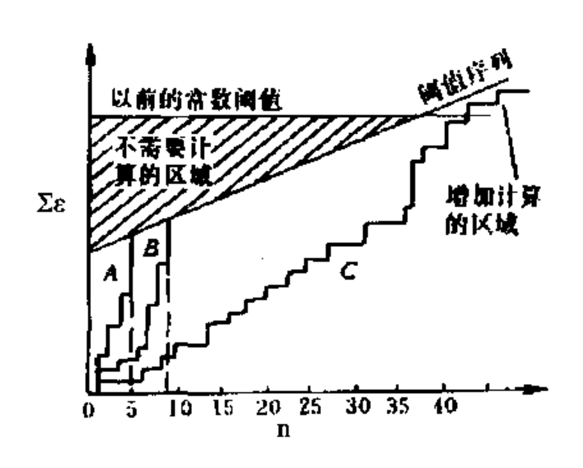


图 9－19 用单调增加阈值序列的情形

SSDA 方法比 FFT 的相关算法快 50 倍，是较受人重视的一种算法。对于二值图，SSDA 方法还可进一步简化。这时模板与对应于图中的成对象点的差值为

$$|S^{ij}(m,n)-T(m,n)|=\overline{S}^{ij}T-\overline{T}S^{ij}$$

$$=S^{ij}(m,n)\oplus T(m,n)$$

式中⊕表示异或处理，由此得到

$$D(i,j)=\sum_{m=1}^{M}\sum_{n=1}^{M}|S^{i,j}(m,n)-T(m,n)|$$

$$=\sum_{m=1}^{M}\sum_{n=1}^{M}S^{i,j}(m,n)\oplus T(m,n)\text{。}$$

这常被称为二进制的 Hamming 距离，D 越小子图同模板越相似。

9.3.2 其他快速计算法

下面我们将以地形和地图匹配的技术为例，讨论加快匹配的其他算法，文中提到的实时图是指飞行器实时测量地面所得到的数据图。而飞行器预存在机内的已知地面的数据图叫做基准图，两图的匹配计算可以对地形进行分析，并可以指示出飞行器当前的位置。

由前面讨论可知，任何一种匹配算法的总计算量都是采用的相关算法的计算量与搜索位置数之积，即

总计算量=（相关算法的计算量）×（搜索位置数）

来决定的。因此，为了减少总的计算量，除上面介绍的方法以外我们再介绍几种其他方法。

一、幅度排序相关算法

这种算法由两个步骤组成。第一步把实时图中的各个灰度值按幅度大小排成列的形式，然后对它进行二进制（或二进制）编码，最后，根据二进位排序的诸列，把实时图变换成二进制阵列的一个有序的集合$\{C_n$，n=1，2，……$N\}$，这一过程称为幅度排序的预处理。第二步，顺序地将这些二进制阵列与基准图进行由粗到细的相关，直到确定出匹配点为止。这里，为了说明这种算法的原理，举一个简单的 3X3 实时图的例子。

第一步：预处理。

首先把 3×3 实时图中各个灰度值按大小次序排成一列，并算出各个灰度值在原图中的位置（j，k）。如图 9－22（a）所示。

然后把排序后的灰度幅度值分成数目相等的两组，且幅度大的一组赋值为 1，幅度小的一组赋值为 0。若幅度数为奇数，则中间的那个幅度就规定为×。如图 9－22（b）所示。把每一组分成两半，并同样地赋予 1 值和 0 值，这个过程一直进行到各组划分为一个单元为止，形成二进制排序。

(a)3×3 实时图

j \ k	1	2	3
1	11	17	1
2	15	6	5
3	14	4	10

(b)预处理步骤(排序)

位置 (j k)	幅度	二进制排序 ①	②	③
1,2	17	1	1	1
2,3	15	1	1	0
3,1	14	1	0	1
1,1	11	1	0	0
3,3	10	×	×	×
2,1	6	0	1	1
2,2	5	0	1	0
3,2	4	0	0	1
1,3	1	0	0	0

图 9—22 3×3 实时图的预处理

于是，根据二进制排序的次序①、②、③和各个二进制值及其位置，便可构成 C_1，C_2，C_3 等二进制阵列。如图 9—23 所示。同理，对于一般情况可得$\{C_n,\ n=1,\ 2,\ \cdots\cdots N\}$，此处，$N$ 为二进制排序的分层数。

C_1

1	1	0
0	0	1
1	0	×

C_2

0	1	0
1	1	1
0	0	×

C_3

0	1	0
1	0	0
1	1	×

图 9—23 本例的二进制阵列

第二步：由粗到细的相关过程。

首先，用 C_1 阵列与基准图阵列作如下相关运算，得

$$\varphi_1(u,v)=\sum_{\substack{j,k\\C_1(j,k)=1}}X_{j+u,k+v}-\sum_{\substack{j,k\\C_1(j,k)=0}}X_{j+u,k+v}$$

这个式子意味着，当 C_1 阵列放在基准图的某一搜索位置（u，v）上时，与 C_1 中的 1 值所对应的基准图的像元值之和减去与 C_1 中的 0 值所对应的基准图的像元值之和（与 C_1 中×号所对应的基准图像元则被忽略）。所以 $\varphi_1(u,v)$ 实际上是一比特量化实时图与基准图的积相关函数，它反映了实时图中最粗糙的图像结构的信息（即一种高低表示）与基准图的相关。$\varphi_1(u,v)$ 被曾称为基本的相关面。

在基准图全区域的检索过程中，若设定一个门限值 T_1，舍弃那些 $\varphi_1(u,v)<T_1$ 的点，就可以大大减少下一轮搜索时的试验位置数。

然而，在$\varphi_1(u,v) > T_1$的试验位置上，可以用下式来进行细的相关运算：

$$\varphi_2(u,v)=\varphi_1(u,v)+\frac{1}{2}\{\sum_{\substack{j,k\\C_2(j,k)=1}}X_{j+u,k+v}-\sum_{\substack{j,k\\C_2(j,k)=0}}X_{j+u,k+v}\}$$

同理，为减少有争议的匹配点的数目，设门限值LT_2，并在$\varphi_2(u,v) > T_2$的试验位置上，以C_2为基础进行更细的相关运算

$$\varphi_3(u,v)=\varphi_2(u,v)+\frac{1}{2^2}\{\sum_{\substack{j,k\\C_3(j,k)=1}}X_{j+u,k+v}-\sum_{\substack{j,k\\C_3(j,k)=0}}X_{j+u,k+v}\}$$

再设门限T_3等等，依此类推，可得第n个相关面为

$$\varphi_n(u,v)=\varphi_{n-1}(u,v)+\frac{1}{2^{n-1}}\{\sum_{\substack{j,k\\C(j,k)=1}}X_{j+u,k+v}-\sum_{\substack{j,k\\C(j,k)=0}}X_{j+u,k+v}\}$$

当设门限值为T_n时，若以$\varphi_n(u^*,v^*) > T_n$的位置只有一个，就宣布该位置(u^*,v^*)为匹配位置。

显然，各个门限值有如下的关系

$$T_n > T_{n-1} > \ldots\ldots > T_2 > T_1$$

因此，逐次细化相关的试验位置将越来越少，直到找出匹配位置时为止。利用此方法减少了总的计算量，并提高相关的处理速度。

这种二进制位的幅度排序算法（即 BARC 算法），所需要的加法总计算量为：

$$q=k(M_1-N_1+1)(M_2-N_2+1)N_1N_2$$

其中 k 是一个在 1 和 2 之间的常数，而M_1、M_2和N_1、N_2分别为基准图和实时图的尺寸。

二、FFT 的相关算法

由付立叶分析中的相关定理可知，两个函数在定义域中的卷积等于它们在频域中的乘积，而相关则是卷积的一种特定形式。因此，存在着另一种计算相关函数的方法。虽然这样做在时间上并没有缩短，但快速付立叶变换技术比直接法计算速度提高了一个数量级，因此用 FFT 进行频域相关计算是一种可行的方法.

首先把基准图和实时图进行二级离散付立叶变换（DFT）。对于基准图，有

$$X(u,v)=\frac{1}{M^2}\sum_{j=0}^{M-1}\sum_{k=0}^{M-1}x(j,k)\omega_M^{-uj}\omega_M^{-vk}$$

其中 u 和 V 分别表示 J 和其在方向上的频率变量，且

$$\omega_M \equiv \exp(j\frac{2\pi}{M})$$

并假定基准图的尺寸是 $M \times M$ 维的。实时图的离散傅立叶变换 y（u,v）是用同样的方式计算得到的。然后，由相关定理可以写出离散傅立叶交换 $\phi(u,v)$ 为

$$\phi(u,v) = X(u,v)Y^*(u,v)$$

为此，对 $\phi(u,v)$ 求付立叶反变换，就可以得出空间域中的相关函数 $\phi(j,k)$ 为

$$\phi(j,k) = \sum_{j=0}^{M-1}\sum_{k=0}^{M-1}[X(u,v)*Y^*(u,v)]\omega_M{}^{uj}\omega_M{}^{vk}$$

其中*为共轭运算的符号。

由此可见，相关函数可以通过 DFT 的方法计算出来，而计算 DFT 最有效的方法就是采用 FFT 算法，这算法种在一般的教科书上都可以找到，所以这里略去对它的讨论。

最后根据以上的关系式，我们可以画出 FFT 的相关算法流程图，如图 9－24 所示。显然，这种相关算法可以容易地推广到 FFT 的归一化相关算法。

如果被测试图像的像元素和试验位置数越大的话，那么这种算法在时间上的节省就更大。但要注意，由于傅立叶变换是个周期性函数，因此匹配点会以周期的形式出现，所以在运算时，必须采取其他适当的措施。

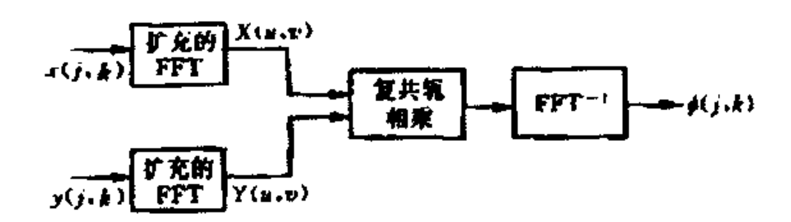


图 9—24 FFT 相关算法

三、分层搜索的序贯判决算法

这种分层搜索算法是基于人们先粗后细寻找事物的习惯而形成的，例如：在世界地图上找北京的位置时，可以先找出中国这个广阔的地域，称其为粗相关。在这个地域内再仔细确定北京的位置，这叫做细相关。所以由这种思想形成的分层搜索算法具有相当高的处理速度。如 BARC 算法一样，它也是由二个步骤组成的。

第一步：预先处理。

首先，对被匹配的图像进行分层预处理。方法是将图 2×2 维的邻区逐个网络地进行平均处理，从而得到一个分辨力较低和维数较小的图像。然后，将此图像再用同样的方法处理后，得到一个分辨力更低和维数更小的图像，依次进行下去。如果一共进行了 K 次分层处理，

那么我们就可以得到 K 个处理后的图像。加上原图像便可构成一组分辨力由高到低，而维数由大到小的图像序列。这种技巧称之为分层预处理。

如果将上述分层技术应用于基准图和实时图，则就可以获得两个这样的图像序列：一个是基准图的，另一个是实时图的。它们分别表示为：

$$X_k:(\frac{M}{2^k}\times\frac{M}{2^k})$$

$$Y_k:(\frac{N}{2^k}\times\frac{N}{2^k})$$

其中 $k=0$，1，……，L，且已假设基准图是 $M\times M$ 维的，而实时图是 $N\times N$ 维的。

因为原图的分辨力最高且维数最大，所以 $k=0$ 的图像 X_0 和 Y_0 具有最高的分辨力，$k=L$ 的图像 X_L 和 Y_L 则具有最低的分辨力。

如前所述，若采用先粗后细的相关方法，则可以很快地找到匹配点。所以，第一次相关搜索是从分辨力最低和维数最小的一对图像 X_K、Y_K（$K=L$）开始的。这时，由于 Y_K 的像元数比较少，加上损失了一部分高频信息，所以在粗相关的过程中，正确截获概率 $P_C{}^k$ 将是不大的。所以，为了提高 $P_C{}^k$ 应设法改善 X_K 和 Y_K 的信噪比。例如：将具有较高分辨力的 Y_{k-1}（或 X_{k-1}）通过低通滤波器以后，再以二倍于它的空间采样间隔进行采样，而得到的 ***YK***（或 X_K）比用直接分层法得到的具有更高的信噪比。因此，相对提高了 $P_C{}^k$。显然，其他各层亦作同样的处理。这种技术称之为分层搜索预处理。

第二步：先粗后细的相关过程。

如前所述，第一次相关是从 Y_k 和 X_k 开始的，为了找到可能的粗匹配位置，应将 Y_k 在 X_k 的所有搜索位置上进行相关，并确定出粗匹配位置 (u^k,v^k)。因为这时 Y_k 和 X_k 的维数最小，所以搜索过程是很快的，但这时 $P_C{}^k$ 值较小，可能产生若干个粗匹配位置。第二次相关是在较高分辨力的图像 Y_{k-1} 和 X_{k-1} 之间进行的。这时，因为已经知道了可能的粗匹配位置，所以 Y_{k-1} 只需要在 X_{k-1} 的一个或若干个粗匹配位置附近进行相关搜索就可以找出一个或少数几个可能性更大的匹配位置 (u^{k-1},v^{k-1})。在上述相关过程中，为了不丢失匹配点，应在粗匹配位置 (u^k,v^k) 附近增加几个补充的试验位置。显然，第三次相关与第二次相关是类似的。如此进行下去，一直到最高分辨力的实时图 X_k 在基准图 X_k 上找到匹配位置时为止。由此可见，整个搜索过程是从最低分辨力到最高分辨力逐层地进行下去的，为了进一步提高处理速度，相关运算常常采用 SSDA 算法，这种技术称为分层搜索的序贯判决算法。

1. 搜索位置数

由上述讨论可知，除了在分辨力最低和尺寸最小的图上作全区域搜索之外，在其他各层的搜索都是在少数几个可能匹配的位置上进行的。因此，当最低分辨力的两图进行相关时，总的搜索位置数为：

$$(\frac{M}{2^L}-\frac{N}{2^L}+1)^2\approx\frac{(M-N)^2}{2^{2L}}$$

而当最高分辨力（$L=0$）的两图相关时，搜索位置数则为

$$(M-N+1)^2\approx(M-N)^2$$

因此，如果不考虑其他各层的搜索位置数（很少）的话，那么分层搜索算法的搜索位置为一般算法的 $1/2^{2L}$，从而提高了处理的速度。

实验表明，当用分层搜索算法时，搜索位置数只有 $K\lg(M\text{-}N\text{-}1)^2$ 个，其中 K=1～2。

2. 分层搜索序贯判决算法的门限序列

这里，假定第 K 次搜索级（即第 K 分层）的低分辨力的图像是由第（K－1）搜索级较高分辨力的图像通过低通滤波器以后，再以二倍于它的间隔采样后得到的。因此，如果低通滤波器的频率特性比较理想的话，那末由采样定理可知，这相当于把低通滤波器的噪声通带减小到 1/2，换言之，从第 K 级搜索到第 K－1 级时，图像噪声增加了 $\sqrt{2}$ 倍。

如果采用 SSDA 算法，及均值一偏差门限序列，则其门限序列公式为

$$T_n^k = (\sqrt{2})^{L^{-k}} (n + g_k \sqrt{n}) r_L$$

其中 T_n^K 表示第 K 搜索级的判决门取序列，k=0，1，2，……L。r_L 是在最低分辨力的 L 级搜索级匹配位置上求得的噪声绝对值的均值，而 g_k 是由搜索级 K 的匹配概率决定的。

应该指出，对于同样的分层搜索技术，若采用不同的滤波预处理和不同的相关算法，就可以形成不同的分层搜索的匹配算法。例如采用对函数的积相关算法，便可以形成分层搜索的对函数匹配算法等。

9.3.3 Visual C++编程实现

模板匹配算法由函数 TemplateMatchDIB()实现。该函数的参数中包括指向两幅图像的指针：lpDIBBits 是指向原 DIB 图像的指针，lpTemplateDIBBits 是指向模板 DIB 图像的指针。程序运行的结果将给出原 DIB 图像中模板 DIB 图像所在的位置。

下面是 TemplateMatchDIB()函数：

```
/*************************************************************************
 *
 * 函数名称:
 *   TemplateMatchDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LPSTR lpTemplateDIBBits  - 指向模版DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *   LONG  lTemplateWidth         - 模板图像宽度（像素数）
 *   LONG  lTemplateHeight        - 模板图像高度（像素数）
 *
 * 返回值:
 *   BOOL                 - 运算成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 * 该函数用于对图像进行模板匹配运算。
 *
 * 要求目标图像为255个灰度值的灰度图像。
 ************************************************************************/

BOOL WINAPI TemplateMatchDIB (LPSTR lpDIBBits, LPSTR lpTemplateDIBBits, LONG lWidth, LONG
```

```
lHeight,
                                        LONG lTemplateWidth,LONG lTemplateHeight)
    {
        // 指向原图像的指针
        LPSTR   lpSrc,lpTemplateSrc;

        // 指向缓存图像的指针
        LPSTR   lpDst;

        // 指向缓存DIB图像的指针
        LPSTR   lpNewDIBBits;
        HLOCAL  hNewDIBBits;

        //循环变量
        long i;
        long j;
        long m;
        long n;

        //中间结果
        double dSigmaST;
        double dSigmaS;
        double dSigmaT;

        //相似性测度
        double R;

        //最大相似性测度
        double MaxR;

        //最大相似性出现位置
        long lMaxWidth;
        long lMaxHeight;

        //像素值
        unsigned char pixel;
        unsigned char templatepixel;

        // 图像每行的字节数
        LONG lLineBytes,lTemplateLineBytes;

        // 暂时分配内存，以保存新图像
        hNewDIBBits = LocalAlloc(LHND, lWidth * lHeight);

        if (hNewDIBBits == NULL)
        {
            // 分配内存失败
            return FALSE;
        }

        // 锁定内存
```

```
lpNewDIBBits = (char * )LocalLock(hNewDIBBits);

// 初始化新分配的内存，设定初始值为255
lpDst = (char *)lpNewDIBBits;
memset(lpDst, (BYTE)255, lWidth * lHeight);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);
lTemplateLineBytes = WIDTHBYTES(lTemplateWidth * 8);

//计算dSigmaT
dSigmaT = 0;

for (n = 0;n < lTemplateHeight ;n++)
{
    for(m = 0;m < lTemplateWidth ;m++)
    {
        // 指向模板图像倒数第j行，第i个像素的指针
        lpTemplateSrc = (char *)lpTemplateDIBBits + lTemplateLineBytes * n + m;
        templatepixel = (unsigned char)*lpTemplateSrc;

        dSigmaT += (double)templatepixel*templatepixel;
    }
}

//找到图像中最大相似性的出现位置
MaxR = 0.0;

for (j = 0;j < lHeight - lTemplateHeight +1 ;j++)
{
    for(i = 0;i < lWidth - lTemplateWidth + 1;i++)
    {
        dSigmaST = 0;
        dSigmaS = 0;

        for (n = 0;n < lTemplateHeight ;n++)
        {
            for(m = 0;m < lTemplateWidth ;m++)
            {
                // 指向原图像倒数第j+n行，第i+m个像素的指针
                lpSrc  = (char *)lpDIBBits + lLineBytes * (j+n) + (i+m);

                // 指向模板图像倒数第n行，第m个像素的指针
                lpTemplateSrc  = (char *)lpTemplateDIBBits + lTemplateLineBytes * n +
m;

                pixel = (unsigned char)*lpSrc;
                templatepixel = (unsigned char)*lpTemplateSrc;

                dSigmaS += (double)pixel*pixel;
                dSigmaST += (double)pixel*templatepixel;
```

```
            }
        }
        //计算相似性
        R = dSigmaST / ( sqrt(dSigmaS)*sqrt(dSigmaT));

        //与最大相似性比较
        if (R > MaxR)
        {
            MaxR = R;
            lMaxWidth = i;
            lMaxHeight = j;
        }
    }
}

//将最大相似性出现区域部分复制到目标图像
for (n = 0;n < lTemplateHeight ;n++)
{
    for(m = 0;m < lTemplateWidth ;m++)
    {
        lpTemplateSrc = (char *)lpTemplateDIBBits + lTemplateLineBytes * n + m;

        lpDst = (char *)lpNewDIBBits + lLineBytes * (n+lMaxHeight) + (m+lMaxWidth);
        *lpDst = *lpTemplateSrc;
    }
}

// 复制图像
memcpy(lpDIBBits, lpNewDIBBits, lWidth * lHeight);

// 释放内存
LocalUnlock(hNewDIBBits);
LocalFree(hNewDIBBits);

// 返回
return TRUE;
}
```

在 ch1_1view.cpp 中添加相应的菜单事件处理程序。和差影法的菜单事件处理程序类似，该程序同样要求用户选择一幅模板图像，用于对原图像进行匹配搜索。

```
void CCh1_1View::OnDetectTemplate()
{
    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR lpDIB;
    LPSTR lpTemplateDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;
    LPSTR lpTemplateDIBBits;
```

```
//图像的宽度与高度
long lWidth,lHeight;

//模板的宽度与高度
long lTemplateWidth;
long lTemplateHeight;

HDIB hTemplateDIB;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());


// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的水平镜像，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的平移！", "系统提示", MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}


// 更改光标形状
BeginWaitCursor();

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

lWidth = ::DIBWidth(lpDIB);
lHeight = ::DIBHeight(lpDIB);

CFileDialog dlg(TRUE,"bmp","*.bmp");
if(dlg.DoModal() == IDOK)
{

    CFile file;
    CFileException fe;

    CString strPathName;

    strPathName = dlg.GetPathName();

    // 打开文件
    VERIFY(file.Open(strPathName, CFile::modeRead | CFile::shareDenyWrite, &fe));
```

```
        // 尝试调用ReadDIBFile()读取图像
        TRY
        {
            hTemplateDIB = ::ReadDIBFile(file);
        }
        CATCH (CFileException, eLoad)
        {
            // 读取失败
            file.Abort();

            // 恢复光标形状
            EndWaitCursor();

            // 报告失败
            //ReportSaveLoadException(strPathName, eLoad,
//              FALSE, AFX_IDP_FAILED_TO_OPEN_DOC);

            // 设置DIB为空
            hTemplateDIB = NULL;

            // 返回
            return;
        }
        END_CATCH

        // 初始化DIB
        //InitDIBData();

        // 判断读取文件是否成功
        if (hTemplateDIB == NULL)
        {
            // 失败，可能非BMP格式
            CString strMsg;
            strMsg = "读取图像时出错！可能是不支持该类型的图像文件！";

            // 提示出错
            MessageBox(strMsg, NULL, MB_ICONINFORMATION | MB_OK);

            // 恢复光标形状
            EndWaitCursor();

            // 返回
            return;
        }
    }
    else
    {
        // 恢复光标形状
        EndWaitCursor();

        return;
```

```
        }
        // 锁定DIB
        lpTemplateDIB = (LPSTR) ::GlobalLock((HGLOBAL) hTemplateDIB);

        // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的模板匹配，其他的可以类推）
        if (::DIBNumColors(lpTemplateDIB) != 256)
        {
            // 提示用户
            MessageBox("目前只支持256色位图！", "系统提示" , MB_ICONINFORMATION | MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) hTemplateDIB);

            // 返回
            return;
        }


        // 更改光标形状
        BeginWaitCursor();

        // 找到DIB图像像素起始位置
        lpTemplateDIBBits = ::FindDIBBits(lpTemplateDIB);

        lTemplateWidth = ::DIBWidth(lpTemplateDIB);
        lTemplateHeight = ::DIBHeight(lpTemplateDIB);

        if(lTemplateHeight > lHeight || lTemplateWidth > lWidth )
        {
            // 提示用户
            MessageBox("模板尺寸大于原图像尺寸！", "系统提示" , MB_ICONINFORMATION | MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) hTemplateDIB);

            // 返回
            return;

        }
        // 调用TemplateMatchDIB()函数进行模板匹配
        if (TemplateMatchDIB(lpDIBBits, lpTemplateDIBBits, lWidth, lHeight,
lTemplateWidth, lTemplateHeight))
        {

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
```

```
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
    ::GlobalUnlock((HGLOBAL) hTemplateDIB);

    // 恢复光标
    EndWaitCursor();

}
```

在头文件 detect.h 中说明了本章各函数的原型：

```
// detect.h

#ifndef _INC_DetectAPI
#define _INC_DetectAPI

// 函数原型
BOOL WINAPI ThresholdDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI AddMinusDIB (LPSTR lpDIBBits, LPSTR lpDIBBitsBK, LONG lWidth, LONG lHeight, bool
bAddMinus);
BOOL WINAPI HprojectDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI VprojectDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI TemplateMatchDIB (LPSTR lpDIBBits, LPSTR lpTemplateDIBBits,
        LONG lWidth, LONG lHeight,LONG lTemplateWidth,LONG lTemplateHeight);

#endif //!_INC_DetectAPI
```

下面给出模板匹配程序运行的一个示例。如图 9－25、图 9－26、图 9－27 所示。



图 9－25 原始图像



图 9－26 待匹配的模板图像

图 9－27 匹配搜索的结果

第十章 图像复原

10.1 引言

图像在形成、传输和记录过程中，由于受多种原因的影响，图像的质量会有所下降，典型表现为图像模糊、失真、有噪声等。这一降质过程称为图像的退化。

图像复原或称图像还原的目的就是尽可能复原被退化图像的本来面目。图像复原和图像增强一样，主要目的是要改善给定图像的质量。但是这两者之间是有着重大区别的。

首先，图像复原试图利用退化现象的某种先验知识（即退化模型），把已经退化了的图像加以重建和复原。引起图像退化的原因很多，有大气湍流效应、传感器特性的非线性、光学系统的像差、成像设备与物体之间的相对运动等等。图像复原需要弄清退化的原因，建立相应的数学模型，并沿着图像降质的逆过程复原图像。而图像增强技术则对图像退化或降质的过程不建立或很少建立模型。

其次，图像复原技术要明确规定质量标准，以便对希望得到的结果做出最佳的评估。而增强技术则主要是一种心理感受过程，它很少涉及客观和统一的评价标准。

由于图像复原过程的特殊性，可以根据不同的退化模型、质量评价标准，导出多种的复原方法。

在 10.1 节中，我们首先介绍图像退化的一般模型。本章所介绍的图像复原方法都基于线性的、空间不变的退化模型。根据所采用的评价准则不同，图像复原的方法又可分为线性代数方法和非线性方法。10.2－10.3 节主要涉及利用矩阵代数的方法对两类复原问题进行研究。在 10.4 节中讨论了几种非线性复原的方法。

图像复原的一个重要特点是需要有关退化过程的先验知识，以及如何对噪声建模的问题。反映在滤波器设计中，即是要求得到点扩展函数(PSF)和噪声模型。在 10.6 节介绍几种典型图像退化过程的点扩展函数，10.7 节则讨论有关噪声模型的问题。

图像退化的一般模型示于图 10－1。原始图像 $f(x,y)$经过一个算子或系统 $H(x,y)$作用后，和加性噪声 $n(x,y)$相叠加，形成退化后的图像 $g(x,y)$，即实际得到的图像。这一过程的数学表达式为：

$$g(x,y)=H[f(x,y)]+n(x,y)$$

$H[*]$可理解为综合所有退化因素的函数。当 $H[*]$是线性算子,即满足：

$$H[k_1 f_1(x,y)+k_2 f_2(x,y)]=k_1 H[f_1(x,y)]+k_2 H[f_2(x,y)]$$ 时(k_1，k_2是常数)

$$g(x,y)=H[f(x,y)]=H[\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)\delta(x-\alpha,y-\beta)\mathrm{d}\alpha\mathrm{d}\beta]$$
$$=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}H[f(\alpha,\beta)\delta(x-\alpha,y-\beta)]\mathrm{d}\alpha\mathrm{d}\beta$$
$$=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)H[\delta(x-\alpha,y-\beta)]\mathrm{d}\alpha\mathrm{d}\beta$$
$$=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)h(x,y;\alpha,\beta)]\mathrm{d}\alpha\mathrm{d}\beta$$

其中，$h(x,y;\alpha,\beta)=H[\delta(x-\alpha,y-\beta)]$ 称为 $H(x,y)$的冲激响应。在图像形成的光学过程中，冲激为一光点。因而又将 $h(x,y;\alpha,\beta)$ 称为退化过程的点扩展函数(PSF)。

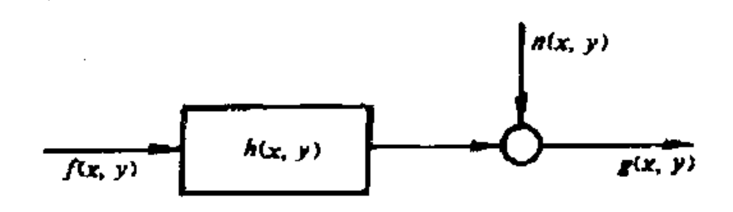


图 10－1 图像退化的一般模型

如果 $H(x,y)$同时又是移不变的，即 $H[*]$满足

$h(x-\alpha,y-\beta)=g(x-\alpha,y-\beta)$ 则 $h(x,y;\alpha,\beta)=H[\delta(x-\alpha,y-\beta)]$

因而

$$g(x,y)=\int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty}f(\alpha,\beta)h(x-\alpha,y-\beta)\mathrm{d}\alpha\mathrm{d}\beta+n(x,y)$$
$$=f(x,y)*h(x,y)+n(x,y) \qquad (10-1)$$

所以，图像复原过程就可看成已知 $g(x,y)$和有关 $h(x,y)$、$n(x,y)$的一些先验知识，求出$f(x,y)$。式 10－1 也是在下面几节中我们推导图像复原算法时采用的退化模型。

在离散情况下，式 10－1 可改写为：

$$g=Hf+n \qquad (10-2)$$

其中

$$g=Vec[g_e(x,y)](x=0,1,\cdots,M-1;y=0,1,\cdots,N-1)$$
$$f=Vec[f_e(x,y)](x=0,1,\cdots,M-1;y=0,1,\cdots,N-1)$$
$$n=Vec[n_e(x,y)](x=0,1,\cdots,M-1;y=0,1,\cdots,N-1)$$

其中

$$g_e(x,y)=f_g(x,y)+h_g(x,y)+n_g(x,y)$$

而 Vec[*]是矩阵代数中将矩阵拉伸为向量的算子。例如

$$Vec[\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}] = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix}$$

而 $f_e(x,y)$、$h_e(x,y)$、$n_e(x,y)$分别是 $f(x,y)$、$h(x,y)$、$n(x,y)$的延拓函数，定义为

如果 $0 \le x \le A-1$ 并且 $0 \le y \le B-1$，则 $f_e(x,y) = f(x,y)$

否则 $f_e(x,y) = 0$

如果 $0 \le x \le C-1$ 并且 $0 \le y \le D-1$，则 $h_e(x,y) = h(x,y)$

否则 $h_e(x,y) = 0$

如果 $0 \le x \le A-1$ 并且 $0 \le y \le B-1$，则 $n_e(x,y) = n(x,y)$

否则 $n_e(x,y) = 0$

其中，A、B、C、D 分别是 $f(x, y)$和 $h(x, y)$的维数。M＝A＋C－1;N＝B＋D－1

H 是一个 $MN \times MN$ 维矩阵：

$$H = \begin{bmatrix} H_0 & H_{M-1} & \cdots & H_1 \\ H_1 & H_0 & \cdots & H_2 \\ \cdots & \cdots & \cdots & \cdots \\ H_{M-1} & H_{M-2} & \cdots & H_0 \end{bmatrix}$$

每个 H_j 都是一个 N×N 的矩阵，定义为：

$$H_j = \begin{bmatrix} h_e(j,0) & h_e(j,N-1) & \cdots & h_e(j,1) \\ h_e(j,1) & h_e(j,0) & \cdots & h_e(j,2) \\ \cdots & \cdots & \cdots & \cdots \\ h_e(j,N-1) & h_e(j,N-2) & \cdots & h_e(j,0) \end{bmatrix}$$

矩阵方程 10－2 看似简单，但实际上对于大的图像，直接求解 f 各元素几乎是不可能的。如果 M＝N＝512．则 H 的大小为 262144* 262144。求解 f 则需求解 262144 个方程组。

由于 H 是分块循环矩阵，可以证明 H 可对角化，即：

$$H = WDW^{-1}$$

W 阵的大小为 $MN \times MN$，由 M^2 个大小为 $N \times N$ 的部分组成，W 的第 im 个部分定义为：

$$W(i,m)=\exp[j\frac{2\pi}{M}im]W_N$$

式中 i，$m=0$，1，2,…,M-1。W_N 为 $N\times N$ 的矩阵：

$$W_N(k,n)=\exp[j\frac{2\pi}{N}kn]$$

式中 k，$n=0$，1，2,…,N-1。对任意形如 H 的分块循环矩阵，W 都可使其对角化。

D 是对角阵，其对角元素与 $h_e(x,y)$的傅立叶变换有关。即如果

$$H(u,v)=\frac{1}{MN}\sum\sum h_e(x,y)\exp[-j2\pi(ux/M+vy/N)]$$

则 D 的 MN 个对角线元素按下面的形式，第一组 N 个元素为 $H(0,0)$、$H(0,1)$、…$H(0,N-1)$；第二组为 $H(1,0)$、$H(1,1)$、…$H(1,N-1)$；依此类推，最后的 N 个对角线元素为 $H(M-1,0)$、$H(M-1,1)$、…$H(M-1,N-1)$。由上述元素组成的整个矩阵再乘以 MN，得到 D，即

$$D(k,i)=\begin{cases}MNH([k/N],k \bmod N), i=k\\0, i\neq k\end{cases}$$

式中[p]用来表示不超过 p 的最大整数。并且 $k \bmod N$ 是以 N 除 k 所得到的余数。进一步有

$$g=Hf+n=WDW^{-1}f+n$$
$$\Rightarrow W^{-1}g=DW^{-1}f+W^{-1}n$$

可以证明，对任意的 s(x，y)有

$$W^{-1}Vec[s(x,y)]=Vec[S(u,v)]=Vec\left[\frac{1}{MN}\sum\sum s(x,y)\exp[-i2\pi(ux/M+vy/N)]\right]$$

所以

$$W^{-1}g=Vec[G(u,v)]$$
$$W^{-1}f=Vec[F(u,v)]$$
$$W^{-1}n=Vec[N(u,v)]$$

因而有

$$G(u,v)=NMH(u,v)F(u,v)+N(u,v) \qquad (10-3)$$

式 10－3 是我们在本章进行图像复原的基础。

10.2 逆滤波器方法——非约束复原

10.2.1 逆滤波器方法

由式(5. 1)可得退化模型中的噪声项为：

$$n = g - Hf$$

当对 n 的统计特性一无所知时，有意义的准则是寻找 f，使 $J(\hat{f}) = \left\|g - H\hat{f}\right\|^2 = \|n\|^2$ 最小，即寻找一个均匀意义下最近似于 g 的 Hf。式中，

$$\|n\|^2 = n^T n, \left\|g - H\hat{f}\right\|^2 = (g - H\hat{f})^T (g - H\hat{f})$$

由极值条件：

$$\frac{\partial \|n\|^2}{\partial \hat{f}} = 0 \Rightarrow H^T (g - H\hat{f}) = 0$$

求出 f

$$\hat{f} = (H^T H)^{-1} H^T g$$

在 M＝N 的情况下，假设 H^{-1} 存在，我们有

$$\hat{f} = H^{-1}(H^{-1})^T H^T g = H^{-1} g$$

可得

$$\hat{f} = (WDW^{-1})^{-1} g = (WD^{-1}W^{-1}) g \Rightarrow W^{-1}\hat{f} = D^{-1}W^{-1}g$$

在根据上节中给出的 W^1 的性质，我们有

$$\hat{F}(u,v) = \frac{G(u,v)}{N^2 H(u,v)} \qquad (10-4)$$

可见，如果知道 $g(x,y)$和 $h(x, y)$，也就知道了 $G(u, v)$和 $H(u, v)$。根据上式即可得出 $F(u, v)$，再经过反付立叶变换就能求出 $f(x,y)$。这种图像复原方法称为逆滤波器复原方法。

如若 $H(u, v)$在 uv 平面上的某些区域等于 0 或变得非常小，那么根据式 10－4 复原就会出现病态性质，即 $F(u, v)$在 $H(u, v)$的零点附近变化剧烈。如果还存在噪声，则后果更加严重。根据式 10－3

$$\hat{F}(u,v) = \frac{G(u,v)}{N^2 H(u,v)} = \frac{N^2 F(u,v) H(u,v) + N(u,v)}{N^2 H(u,v)} = F(u,v) + \frac{N(u,v)}{N^2 H(u,v)}$$

那么在 $F(u,v) \le \dfrac{N(u,v)}{N^2H(u,v)}$ 的 uv 区域中(这是非常有可能的，因为 $H(u,v)$ 随 $\sqrt{u^2+v^2}$ 急剧下降，而 $N(u,v)$ 则几乎是不下降的)噪声将掩盖一切，造成复原出的图像面目全非。解决这两种病态性质可用下面的方法：一是在计算 $\hat{F}(u,v) = \dfrac{G(u,v)}{N^2H(u,v)}$ 时，在 $H(u,v)$ 的零点上不做计算，或直接对 $H(u,v)$ 进行修改，即仔细设置 $H(u,v)=0$ 的频谱点附近 $H^{1}(u,v)$ 的值；另外是在原点的有限的邻域内进行修改，以避免小数值的 $H(u,v)$。即选择一个低通滤波器：

$$H_1(u,v) = \begin{Bmatrix} 1\sqrt{u^2+v^2} \le D_0 \\ 0\sqrt{u^2+v^2} > D_0 \end{Bmatrix}$$

并进行如下的反向滤波还原

$$\hat{F}(u,v) = \frac{G(u,v)H_1(u,v)}{H(u,v)}$$

为避免振铃影响，还可选择平滑的低通滤波器如 Butterworth 滤波器等等代替 $H_1(u,v)$。

在导出时，假设 H^1 存在。由极值条件我们得到

$$H^T g = H^T H\hat{f} \Rightarrow$$

$$WD^*W^{-1}g = WD^*W^{-1}WDW^{-1}\hat{f}$$

$$D^*Vec[G(u,v)] = D^*DVec[\hat{F}(u,v)]$$

$$\hat{F}(u,v) = \begin{Bmatrix} \dfrac{G(u,v)}{N^2H(u,v)} D(u,v) \ne 0 H(u,v) \ne 0 \\ D(u,v) = 0 H(u,v) = 0 \end{Bmatrix}$$

*为共轭转置。

实验证明，当变质图像的信噪比较高（例如信噪比 SNR1000 或更高，而且轻度变质）时逆滤波复原方法可以获得较好的效果。这种情况可以用图 10－2 说明。由图可见，变质图像的频带|G|被限制在较小范围内，而且在这个范围内，传输函数|H|没有零点而且不是太小，这时可以在变质图像存在的频谱范围内用逆滤波复原方法，在此范围之外，可认为图像为零。

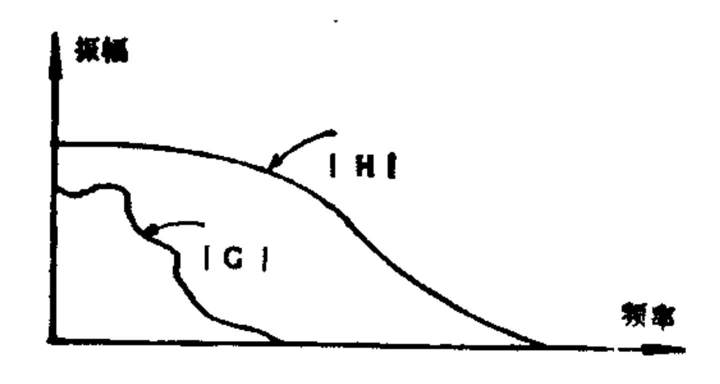


图 10－2 退化图像频谱的分布

通常系统存在噪声而且传输函数存在零点时采用逆滤波方法将会碰到困难，但是下一节将介绍的维纳滤波方法可以解决这一问题。

10.2.2 Visual C++编程实现

根据式（10－4），我们可以编程实现图像的逆滤波方法复原。图像复原需要假定点扩展函数 H 是已知的，所以应首先用一个已知的点扩展函数对图像进行模糊操作，生成一幅待复原的图像：

$$g = h * f$$

其中 $h = \frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\\1 & 1 & 1 & 1 & 1\end{bmatrix}$，即用一个 5×5 的模板对原始图像进行平滑模糊操作。

上式卷积实现的具体过程为

$$g = \Im^{-1}(\Im(h) \bullet \Im(f))$$

即先求原始图像和点扩展函数的付立叶变换，在频域相乘再求其反变换。其中的付立叶变换和付立叶反变换利用 n 维付立叶变换函数 fourn(double * data, unsigned long nn[], int ndim, int isign)实现，其中的参数 data 为图像数组，nn 为各维的长度（要求为 2 的整数幂），ndim 为付立叶变换的维数，isign 指明是付立叶变换(isign＝1)或付立叶反变换(isign＝-1)。

而逆滤波的具体实现过程为

$$\hat{f} = \Im^{-1}(\frac{G}{H}) = \Im^{-1}(\frac{\Im(g)}{\Im(h)})$$

其中的付立叶变换和反变换同样利用函数 fourn()实现，对图像进行付立叶变换要求图像的长度和宽度必须为 2 的整数次幂。

首先利用资源编辑器加入如下的菜单。如图 10－3 所示。

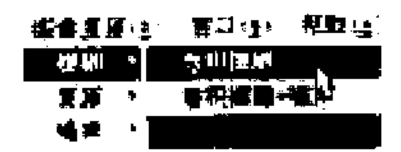


图 10-3 图像复原操作菜单

实现卷积模糊操作的函数 BlurDIB()如下：

```
// ************************************************************************
//  文件名：restore.cpp
//
//  图像复原API函数库：
//
//  BlurDIB()          - 图像模糊
//  InverseDIB()       - 图像逆滤波
//  NoiseBlurDIB()     - 图像模糊加噪
//  WienerDIB()        - 图像维纳滤波
//  RandomNoiseDIB()   - 图像中加入随机噪声
//  SaltNoiseDIB()     - 图像中加入椒盐噪声
//  fourn()            - n维FFT
//
// ************************************************************************

#include "stdafx.h"
#include "restore.h"
#include "DIBAPI.h"

#include <math.h>
#include <direct.h>

#define SWAP(a,b) tempr=(a);(a)=(b);(b)=tempr

/*************************************************************************
 *
 * 函数名称：
 *   BlurDIB()
 *
 * 参数：
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值：
 *   BOOL               - 模糊操作成功返回TRUE，否则返回FALSE。
 *
 * 说明：
 *   该函数用来对DIB图像进行模糊操作。
 *
 ************************************************************************/

BOOL WINAPI BlurDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
```

```
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;

    //用于做FFT的数组
    double *fftSrc,*fftKernel;
    //二维FFT的长度和宽度
    unsigned long nn[3];
    //图像归一化因子
    double MaxNum;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    double dPower = log((double)lLineBytes)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }
    dPower = log((double)lHeight)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }

    fftSrc = new double [lHeight*lLineBytes*2+1];
    fftKernel = new double [lHeight*lLineBytes*2+1];

    nn[1] = lHeight;
    nn[2] = lLineBytes;

    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            pixel = (unsigned char)*lpSrc;

            fftSrc[(2*lLineBytes)*j + 2*i + 1] = (double)pixel;
```

```
            fftSrc[(2*lLineBytes)*j + 2*i + 2] = 0.0;

            if(i < 5 && j < 5)
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 1/25.0;
            }
            else
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 0.0;
            }
            fftKernel[(2*lLineBytes)*j + 2*i + 2] = 0.0;
        }
    }

    //对原图像进行FFT
    fourn(fftSrc,nn,2,1);
    //对卷积核图像进行FFT
    fourn(fftKernel,nn,2,1);

    //频域相乘
    for (i = 1;i <lHeight*lLineBytes*2;i+=2)
    {
        fftSrc[i] = fftSrc[i] * fftKernel[i] - fftSrc[i+1] * fftKernel[i+1];
        fftSrc[i+1] = fftSrc[i] * fftKernel[i+1] + fftSrc[i+1] * fftKernel[i];
    }

    //对结果图像进行反FFT
    fourn(fftSrc,nn,2,-1);

    //确定归一化因子
    MaxNum = 0.0;
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            fftSrc[(2*lLineBytes)*j + 2*i + 1] =
                sqrt(fftSrc[(2*lLineBytes)*j + 2*i + 1] * fftSrc[(2*lLineBytes)*j + 2*i +
1]\
                        +fftSrc[(2*lLineBytes)*j + 2*i + 2] * fftSrc[(2*lLineBytes)*j + 2*i
+ 2]);
            if( MaxNum < fftSrc[(2*lLineBytes)*j + 2*i + 1])
                MaxNum = fftSrc[(2*lLineBytes)*j + 2*i + 1];
        }
    }

    //转换为图像
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            // 指向原图像倒数第j行，第i个像素的指针
```

```
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            *lpSrc = (unsigned char) (fftSrc[(2*lLineBytes)*j + 2*i + 1]*255.0/MaxNum);
        }
    }
    delete fftSrc;
    delete fftKernel;
    // 返回
    return true;
}
```

头文件 restore.h 如下：

```
// restore.h

#ifndef _INC_RestoreAPI
#define _INC_RestoreAPI

// 函数原型
BOOL fourn(double * data, unsigned long nn[], int ndim, int isign);
BOOL WINAPI BlurDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI RestoreDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI NoiseBlurDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI WienerDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI RandomNoiseDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);
BOOL WINAPI SaltNoiseDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight);

#endif //!_INC_RestoreAPI
```

对应的菜单事件处理函数：

```
void CCh1_1View::OnRestoreBlur()
{
    //图像模糊操作，生成一幅待复原的图像

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的模糊操作，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);
```

```
        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用BlurlDIB()函数对DIB进行模糊处理
    if (BlurDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败或图像尺寸不符合要求！", "系统提示", MB_ICONINFORMATION |
MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

图像的逆滤波复原操作由函数 RestoreDIB()实现：

```
/*************************************************************************
 *
 * 函数名称:
 *   RestoreDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits      - 指向原DIB图像指针
 *   LONG  lWidth         - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight        - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL                 - 复原操作成功返回TRUE，否则返回FALSE。
 *
```

```
 * 说明:
 *   该函数用来对BlurDIB()生成的DIB图像进行复原操作。
 *
 *************************************************************************/

BOOL WINAPI RestoreDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;

    //用于做FFT的数组
    double *fftSrc,*fftKernel;
    double a,b,c,d;
    //二维FFT的长度和宽度
    unsigned long nn[3];
    //图像归一化因子
    double MaxNum;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    double dPower = log((double)lLineBytes)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }
    dPower = log((double)lHeight)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }

    fftSrc = new double [lHeight*lLineBytes*2+1];
    fftKernel = new double [lHeight*lLineBytes*2+1];

    nn[1] = lHeight;
    nn[2] = lLineBytes;

    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
```

```
        {
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            pixel = (unsigned char)*lpSrc;

            fftSrc[(2*lLineBytes)*j + 2*i + 1] = (double)pixel;
            fftSrc[(2*lLineBytes)*j + 2*i + 2] = 0.0;

            if(i < 5 && j < 5)
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 1/25.0;
            }
            else
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 0.0;
            }
            fftKernel[(2*lLineBytes)*j + 2*i + 2] = 0.0;
        }
    }

    //对原图像进行FFT
    fourn(fftSrc,nn,2,1);
    //对卷积核图像进行FFT
    fourn(fftKernel,nn,2,1);

    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            a = fftSrc[(2*lLineBytes)*j + 2*i + 1];
            b = fftSrc[(2*lLineBytes)*j + 2*i + 2];
            c = fftKernel[(2*lLineBytes)*j + 2*i + 1];
            d = fftKernel[(2*lLineBytes)*j + 2*i + 2];
            if (c*c + d*d > 1e-3)
            {
                fftSrc[(2*lLineBytes)*j + 2*i + 1] = ( a*c + b*d ) / ( c*c + d*d );
                fftSrc[(2*lLineBytes)*j + 2*i + 2] = ( b*c - a*d ) / ( c*c + d*d );
            }
        }
    }

    //对结果图像进行反FFT
    fourn(fftSrc,nn,2,-1);

    //确定归一化因子
    MaxNum = 0.0;
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
```

```
            fftSrc[(2*lLineBytes)*j + 2*i + 1] =
                sqrt(fftSrc[(2*lLineBytes)*j + 2*i + 1] * fftSrc[(2*lLineBytes)*j + 2*i +
1]\
                        +fftSrc[(2*lLineBytes)*j + 2*i + 2] * fftSrc[(2*lLineBytes)*j + 2*i
+ 2]);
            if( MaxNum < fftSrc[(2*lLineBytes)*j + 2*i + 1])
                MaxNum = fftSrc[(2*lLineBytes)*j + 2*i + 1];
        }
    }


    //转换为图像
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            *lpSrc = (unsigned char) (fftSrc[(2*lLineBytes)*j + 2*i + 1]*255.0/MaxNum);
        }
    }
    delete fftSrc;
    delete fftKernel;
    // 返回
    return true;
}
```

在 ch1_1view.cpp 中对应的菜单事件处理函数：

```
void CCh1_1View::OnRestoreInverse()
{
    //图像逆滤波复原操作

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的复原操作，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！","系统提示",MB_ICONINFORMATION | MB_OK);

        // 解除锁定
```

```
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用RestoreDIB()函数对DIB进行复原
    if (RestoreDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败或图像尺寸不符合要求！", "系统提示", MB_ICONINFORMATION |
MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

用于二维付立叶变换和付立叶反变换的函数 **fourn()**如下：

```
BOOL fourn(double * data, unsigned long nn[], int ndim, int isign)
{
    int idim;
    unsigned long i1,i2,i3,i2rev,i3rev,ip1,ip2,ip3,ifp1,ifp2;
    unsigned long ibit,k1,k2,n,nprev,nrem,ntot;
    double tempi,tempr;
    double theta,wi,wpi,wpr,wr,wtemp;

    for (ntot=1,idim=1;idim<=ndim;idim++)
        ntot *= nn[idim];
    nprev=1;
    for (idim=ndim;idim>=1;idim--) {
        n=nn[idim];
```

```
    nrem=ntot/(n*nprev);
    ip1=nprev << 1;
    ip2=ip1*n;
    ip3=ip2*nrem;
    i2rev=1;
    for (i2=1;i2<=ip2;i2+=ip1) {
        if (i2 < i2rev) {
            for (i1=i2;i1<=i2+ip1-2;i1+=2) {
                for (i3=i1;i3<=ip3;i3+=ip2) {
                    i3rev=i2rev+i3-i2;
                    SWAP(data[i3],data[i3rev]);
                    SWAP(data[i3+1],data[i3rev+1]);
                }
            }
        }
        ibit=ip2 >> 1;
        while (ibit >= ip1 && i2rev > ibit) {
            i2rev -= ibit;
            ibit >>= 1;
        }
        i2rev += ibit;
    }
    ifp1=ip1;
    while (ifp1 < ip2) {
        ifp2=ifp1 << 1;
        theta=isign*6.28318530717959/(ifp2/ip1);
        wtemp=sin(0.5*theta);
        wpr = -2.0*wtemp*wtemp;
        wpi=sin(theta);
        wr=1.0;
        wi=0.0;
        for (i3=1;i3<=ifp1;i3+=ip1) {
            for (i1=i3;i1<=i3+ip1-2;i1+=2) {
                for (i2=i1;i2<=ip3;i2+=ifp2) {
                    k1=i2;
                    k2=k1+ifp1;
                    tempr=wr*data[k2]-wi*data[k2+1];
                    tempi=wr*data[k2+1]+wi*data[k2];
                    data[k2]=data[k1]-tempr;
                    data[k2+1]=data[k1+1]-tempi;
                    data[k1] += tempr;
                    data[k1+1] += tempi;
                }
            }
            wr=(wtemp=wr)*wpr-wi*wpi+wr;
            wi=wi*wpr+wtemp*wpi+wi;
        }
        ifp1=ifp2;
    }
    nprev *= n;
}
```

```
    return true;
}
```

下面给出用该程序对一副图像进行卷积模糊操作和逆滤波复原的示例。如图 10—4、图 10—5、图 10—6 所示。



图 10—4 原始图像



图 10—5 卷积模糊后的结果图像



图 10—6 对图 10—5 进行逆滤波复原后的结果图像

10.3 最小二乘类约束复原

在 10.2 节中介绍的反向滤波法中，对 f 复原出的图像根本不做任何约束和规定。这就是称反向滤波法为非约束复原的原因之一。另外，我们有时对 f 做一些特殊的规定，这反映在 f 必在满足式的同时，还必须使某个指标达到极值。

本节中考虑如下形式的指标：$\|Q\hat{f}\|^2$，其中 Q 为线性算子。$\|Q\hat{f}\|^2 = (Q\hat{f})^T Q\hat{f} = \hat{f}Q^T Q\hat{f}$ 。

通过拉格朗日乘子法，我们知道使$\left\|Q\hat{f}\right\|^2$最小的$f$可由下式求得：

$$J(\hat{f})=\left\|Q\hat{f}\right\|^2+\alpha\left(\left\|g-H\hat{f}\right\|^2+\left\|n\right\|^2\right)$$

$$\frac{\partial J(\hat{f})}{\partial f}=0$$

α为拉格朗日乘子。

由极值条件，我们有：

$$2Q^TQ\hat{f}=2\alpha H^T(g-H\hat{f})$$

$$\hat{f}=(H^TH+\gamma Q^TQ)^{-1}H^Tg \qquad (10-5)$$

其中$\gamma=1/\alpha$，它必须使约束条件式 10－2 得到满足。

下面讨论两种重要的最小二乘法约束还原，它们分别是通过选择不同的Q而得到的。

10.3.1 维纳滤波方法

设R_f和R_n为f和M的相关矩阵，其定义为

$$R_f=E\{ff^T\}$$

$$R_n=E\{nn^T\}$$

E{}代表数学期望运算。

易见Rr和R_n为实对称矩阵，所以可以定义$Q^TQ=R_f^{-1}R_n$。代入式可得

$$\hat{f}=(H^TH+\gamma R_f^{-1}R_n)^{-1}H^Tg \qquad (10-6)$$

假设任两个像素之间的相关是像素之间距离的函数而不是位置的函数，则R_f和R_n可近似为分块循环阵。可采用 10.1 节中介绍的W矩阵进行对角化。设

$$R_f=WAW^{-1}$$

$$R_n=WBW^{-1}$$

A和B与R_f、R_n的傅立叶变换有关,即与谱密度$S_f(u,v)$、$S_n(u,v)$有关：

$$A(k,i)=\begin{cases}MNS_f([k/N],k\bmod N) & i=k\\ 0 & i\neq k\end{cases}$$

$$B(k,i)=\begin{cases}MNS_n([k/N],k\bmod N) & i=k\\ 0 & i\neq k\end{cases}$$

将上述关系代入式(5．6)中可得：

$\hat{f}=(H^T H+\gamma R_f^{-1}R_n)^{-1}H^T g \Rightarrow$

$W^{-1}\hat{f}=(D\cdot D+\gamma A^{-1}B)^{-1}D\cdot W^{-1}g$

所以我们有(假设 $M=N$)

$$\hat{F}(u,v)=\frac{N^2 H^*(u,v)}{N^4\left|H(u,v)\right|^2+\dfrac{\gamma S_n(u,v)}{S_f(u,v)}}G(u,v)$$

这种形式的图像复原称为参变维纳滤波法。这种方法对噪声放大有自动抑制作用。当 $H(u,v)$在某处为零时，由于存在 $\dfrac{S_n(u,v)}{S_f(u,v)}$ 项，就不会出现被零除的情形；同时分子含有 $H(u,v)^*$ 项，在任何 $H(u,v)=0$ 处，滤波器的增益恒等于 0。另外，如果在某一频谱区信噪比相当高，即 $S_n(u,v)<<S_f(u,v)$ 时，滤波器的效果也趋向于反向滤波方法。反之，对信噪比很小的区域，即 $S_n(u,v)>>S_f(u,v)$ 时，滤波器趋向于无反应。这表明维纳滤波避免了在反向滤波中出现的对噪声的过多放大作用。

在使用参变维纳滤波法时，$H(u,v)$由点扩展函数确定，而 $S_f(u,v)$ 和 $S_n(u,v)$ 分别用下面的方法确定：

假定噪声是白噪声，则 $S_n(u,v)$ 为常数。故可通过计算一幅噪声图像的功率谱求得。对 $S_f(u,v)$ 来说，由于 $S_g(u,v)=\left|H(u,v)\right|^2 S_f(u,v)+S_n(u,v)$ 所以可用

$\dfrac{S_g(u,v)-S_n(u,v)}{\left|H(u,v)\right|^2}$ 来估计 $S_f(u,v)$。

另一个常用的方法是使用如下的滤波器：

$\hat{F}(u,v)=\dfrac{H^*(u,v)}{N^2\left|H(u,v)\right|^2+K}G(u,v)$，其中又是常数。

关于 γ，我们在前面已经指出，它必须使约束方程式 10－2 得到满足。然而可以证明，令 $\gamma=1$ 时，我们得到的滤波器在统计的意义上是最佳的，它使 $E\{[f(x,y)-\hat{f}(x,y)]^2\}$ 最小。这种滤波器就是所谓的维纳滤波器。

维纳滤波图像复原方法在大多数实际情况下都可得到满意的结果。但是当信噪比很低的情况下，复原结果还不能令人满意，这可能是由于以下一些因素造成的。

1. 维纳滤波器是假设线性系统。但实际上，图像的记录和评价图像的人类视觉系统往

往都是非线性的。

2. 维纳滤波器是根据最小均方误差准则设计的滤波器，这个准则不见得与人类视觉判决准则相符合。
3. 维纳滤波是基于平稳随机过程的模型，实际存在的千奇百怪的图像并不一定都符合这个模型。另外，维纳滤波只利用了图像的协方差信息，可能还有大量的有用信息没有充分利用。

总之，如果满足平稳随机过程的模型和变质系统是线性的两个条件，那么维纳滤波将会取得较好的复原效果。

10.3.2 约束最小平方滤波

在上一节中推导参变维纳滤波器时，我们隐含了一个重要的假设：即f和$f^\wedge$是随机变量。它们的图像集形成了随机过程，而我们认为这一过程是平稳的随机过程，这意味着参变维纳滤波器得到的结果在图像平均意义下最佳。本节中讨论另一种准则函数，它导出的复原技术使得我们能够对每一幅图像确定一种最优评判标准。

我们知道，使用逆滤波器一类的方法进行图像复原时，由于 $H(u,v)$的病态性质，导致在零点附近数值起伏过大，使复原后的图像产生了多余的噪声和边沿。通过选择合理的 Q，并对$\|Qf\|^2$进行优化，我们可将这种图像的不平滑性降至最小。

考虑 Laplacian 算子$\nabla^2 f=(\dfrac{\partial^2}{\partial x^2}+\dfrac{\partial^2}{\partial y^2})f$。它具有突出边缘的作用。而$\iint \nabla^2 f\mathrm{d}x\mathrm{d}y$则刻划了整幅图像的平滑性。因此可以考虑将其作为图像复原时的约束。

下面我们考虑如何将其表达为$\|Qf\|^2$的形式，从而可使用式 10－5。

在离散的情况下，∇^2可用下面的差分运算来近似：

$$\frac{\partial^2 f(x,y)}{\partial x^2}+\frac{\partial^2 f(x,y)}{\partial y^2}=f(x+1,y)-2f(x,y)+f(x-1,y)+f(x,y+1)-2f(x,y)$$

$$+f(x,y-1)=f(x+1,y)+f(x-1,y)+f(x,y+1)+f(x,y-1)-4f(x,y)$$

利用$f(x,\ y)$与下面的算子进行卷积可实现上面的运算：

$$p(x,y)=\begin{bmatrix}0 & 1 & 0\\ 1 & -4 & 1\\ 0 & 1 & 0\end{bmatrix}$$

当然，卷积前 $p(x,\ y)$必须延拓。设延拓后的函数为 $p_e(x,\ y)$。建立分块循环矩阵：

$$C=\begin{bmatrix} C_0 & C_{M-1} & C_{M-2} & \cdots & C_1 \\ C_1 & C_0 & C_{M-1} & \cdots & C_2 \\ C_2 & C_1 & C_0 & \cdots & C_3 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ C_{M-1} & C_{M-2} & C_{M-3} & \cdots & C_0 \end{bmatrix}$$

式中每个子矩阵 C_j 是由 $p_e(x,\ y)$的第 j 行组成的 $N\times N$ 循环矩阵。即：

$$C_j=\begin{bmatrix} p_e(j,0) & p_e(j,N-1) & \cdots & p_e(j,1) \\ p_e(j,1) & p_e(j,0) & \cdots & p_e(j,2) \\ \cdots & \cdots & \cdots & \cdots \\ p_e(j,N-1) & p_e(j,N-2) & \cdots & p_e(j,0) \end{bmatrix}$$

在上述符号的约定下，$\iint \nabla^2 f dxdy$ 可写为 f^TC^TCf。定义 $Q=C$，则有 $f^TC^TCf=\|Qf\|^2$。

因此我们的问题是在满足式 10－2 约束的条件下，最小化 $\|Qf\|^2$。注意到 C 是分块循环矩阵，因而可用 W 阵进行对角化，设 $C=WEW^{1}$，其中 E 为：

$$E(k,i)=\begin{Bmatrix} MNP([\frac{k}{N}],k \bmod N)i=k \\ 0i\neq k \end{Bmatrix}$$

而 $P(u,\ v)$为 $p_e(x,\ y)$的傅立叶变换。从而

$$\hat{f}=(H^TH+\gamma C^TC)^{-1}H^Tg=(WD\cdot DW^{-1}+\gamma WE\cdot EW^{-1})^{-1}WD\cdot W^{-1}g$$

$$W^{-1}\hat{f}=W^{-1}(WD\cdot DW^{-1}+\gamma WE\cdot EW^{-1})^{-1}WD\cdot W^{-1}g$$

$$=(WD\cdot D+\gamma WE\cdot E)^{-1}WD\cdot W^{-1}g$$

$$=(D\cdot D+\gamma E\cdot E)^{-1}D\cdot W^{-1}g$$

即：

$$\hat{F}(u,v)=\frac{N^2H^*(u,v)}{N^4|H(u,v)|^2+\gamma N^4|P(u,v)|^2}G(u,v)$$

$$=\frac{H^*(u,v)}{N^2|H(u,v)|^2+\gamma N^2|P(u,v)|^2}G(u,v)$$

此滤波器称为最小平方滤波器。

10.3.3 Visual C++编程实现

我们首先利用卷积模糊和加噪操作生成一副待复原的图像：

$$g=h*f+n$$

$$= F^{-1}(F(h) \bullet F(f)) + n$$

根据前面叙述的维纳滤波的过程，对生成的图像进行维纳滤波复原：
由于

$$\hat{F}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + \frac{S_n(u,v)}{S_f(u,v)}} G(u,v)$$

而$S_f(u,v) = \dfrac{S_g(u,v) - S_n(u,v)}{|H(u,v)|^2}$，代入上式，得到：

$$\hat{F}(u,v) = \frac{H^*(u,v)}{|H(u,v)|^2 + \frac{S_n(u,v)}{\frac{S_g(u,v) - S_n(u,v)}{|H(u,v)|^2}}} G(u,v)$$

$$= \frac{H^*(u,v) \bullet G(u,v)}{|H(u,v)|^2 \frac{S_g(u,v)}{S_g(u,v) - S_n(u,v)}}$$

根据这一式子，我们可以编写出实现维纳滤波复原的程序。

对图像进行卷积模糊和加噪操作的函数 NoiseBlurDIB()如下，其中的卷积运算使用下面的点扩展函数

$$h = \frac{1}{5}[1,1,1,1,1]$$

而加噪运算则在图像中加入斜45°方向的网格状噪声（见图10－8）。

```
/*************************************************************************
 *
 * 函数名称:
 *   NoiseBlurDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               - 模糊加噪操作成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对DIB图像进行模糊加噪操作。
 *
 ************************************************************************/
```

```
BOOL WINAPI NoiseBlurDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;

    //用于做FFT的数组
    double *fftSrc,*fftKernel;
    //二维FFT的长度和宽度
    unsigned long nn[3];
    //图像归一化因子
    double MaxNum;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    double dPower = log((double)lLineBytes)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }
    dPower = log((double)lHeight)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }

    fftSrc = new double [lHeight*lLineBytes*2+1];
    fftKernel = new double [lHeight*lLineBytes*2+1];

    nn[1] = lHeight;
    nn[2] = lLineBytes;

    for (j = 0;j < lHeight ;j++)
    {
        i = i;
        for(i = 0;i < lLineBytes ;i++)
        {
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;
```

```
            pixel = (unsigned char)*lpSrc;

            fftSrc[(2*lLineBytes)*j + 2*i + 1] = (double)pixel;
            fftSrc[(2*lLineBytes)*j + 2*i + 2] = 0.0;

            if(i < 5 && j == 0)
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 1/5.0;
            }
            else
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 0.0;
            }
            fftKernel[(2*lLineBytes)*j + 2*i + 2] = 0.0;
        }
    }

    //对原图像进行FFT
    fourn(fftSrc, nn, 2, 1);
    //对卷积核图像进行FFT
    fourn(fftKernel, nn, 2, 1);

    //频域相乘
    for (i = 1;i <lHeight*lLineBytes*2;i+=2)
    {
        fftSrc[i] = fftSrc[i] * fftKernel[i] - fftSrc[i+1] * fftKernel[i+1];
        fftSrc[i+1] = fftSrc[i] * fftKernel[i+1] + fftSrc[i+1] * fftKernel[i];
    }

    //对结果图像进行反FFT
    fourn(fftSrc, nn, 2, -1);

    //确定归一化因子
    MaxNum = 0.0;
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            fftSrc[(2*lLineBytes)*j + 2*i + 1] =
                sqrt(fftSrc[(2*lLineBytes)*j + 2*i + 1] * fftSrc[(2*lLineBytes)*j + 2*i +
1]\
                        +fftSrc[(2*lLineBytes)*j + 2*i + 2] * fftSrc[(2*lLineBytes)*j + 2*i
+ 2]);
            if( MaxNum < fftSrc[(2*lLineBytes)*j + 2*i + 1])
                MaxNum = fftSrc[(2*lLineBytes)*j + 2*i + 1];
        }
    }

    //转换为图像，加噪
    char point;
```

```
        for (j = 0;j < lHeight ;j++)
        {
            for(i = 0;i < lLineBytes ;i++)
            {
                if ( i - j == ((int)((i+j)/8))*8)
                {
                    point = -16;
                }
                else
                {
                    point = 0;
                }

                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

                *lpSrc = (unsigned char) (fftSrc[(2*lLineBytes)*j + 2*i + 1]*255.0/MaxNum +
point);
            }
        }

        delete fftSrc;
        delete fftKernel;
        // 返回
        return true;
    }
```

对应的菜单事件处理函数为 **OnRestoreNoiseblur()**：

```
    void CCh1_1View::OnRestoreNoiseblur()
    {
        //图像模糊操作，生成一幅待复原的图像

        // 获取文档
        CCh1_1Doc* pDoc = GetDocument();

        // 指向DIB的指针
        LPSTR    lpDIB;

        // 指向DIB像素指针
        LPSTR    lpDIBBits;

        // 锁定DIB
        lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

        // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的模糊操作，其他的可以类推）
        if (::DIBNumColors(lpDIB) != 256)
        {
            // 提示用户
            MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

            // 解除锁定
            ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());
```

```
        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用NoiseBlurDIB()函数对DIB进行模糊加噪处理
    if (NoiseBlurDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败或图像尺寸不符合要求！", "系统提示", MB_ICONINFORMATION |
MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

对图像进行维纳滤波复原的函数为 WienerDIB()：

```
/*************************************************************************
 *
 * 函数名称：
 *   WienerDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL                - 维纳滤波复原操作成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对DIB图像进行维纳滤波复原操作。
```

```
 *
 ************************************************************************/

BOOL WINAPI WienerDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    //像素值
    unsigned char pixel;

    // 图像每行的字节数
    LONG lLineBytes;

    //用于做FFT的数组
    double *fftSrc,*fftKernel,*fftNoise;
    double a,b,c,d,e,f,multi;
    //二维FFT的长度和宽度
    unsigned long nn[3];
    //图像归一化因子
    double MaxNum;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    double dPower = log((double)lLineBytes)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }
    dPower = log((double)lHeight)/log(2.0);
    if(dPower != (int) dPower)
    {
        return false;
    }

    fftSrc = new double [lHeight*lLineBytes*2+1];
    fftKernel = new double [lHeight*lLineBytes*2+1];
    fftNoise = new double [lHeight*lLineBytes*2+1];

    nn[1] = lHeight;
    nn[2] = lLineBytes;

    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
```

```
            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            pixel = (unsigned char)*lpSrc;

            fftSrc[(2*lLineBytes)*j + 2*i + 1] = (double)pixel;
            fftSrc[(2*lLineBytes)*j + 2*i + 2] = 0.0;

            if(i < 5 && j == 0)
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 1/5.0;
            }
            else
            {
                fftKernel[(2*lLineBytes)*j + 2*i + 1] = 0.0;
            }
            fftKernel[(2*lLineBytes)*j + 2*i + 2] = 0.0;
            if ( i + j == ((int)((i+j)/8))*8)
            {
                fftNoise [(2*lLineBytes)*j + 2*i + 1]= -16.0;
            }
            else
            {
                fftNoise [(2*lLineBytes)*j + 2*i + 1]= 0.0;
            }
            fftNoise[(2*lLineBytes)*j + 2*i + 2] = 0.0;

        }
    }

    srand((unsigned)time(NULL));

    //对原图像进行FFT
    fourn(fftSrc,nn,2,1);
    //对卷积核图像进行FFT
    fourn(fftKernel,nn,2,1);
    //对噪声图像进行FFT
    fourn(fftNoise,nn,2,1);

    for (i = 1;i <lHeight*lLineBytes*2;i+=2)
    {
            a = fftSrc[i];
            b = fftSrc[i+1];
            c = fftKernel[i];
            d = fftKernel[i+1];
            e = fftNoise[i];
            f = fftNoise[i+1];
            multi = (a*a + b*b)/(a*a + b*b - e*e - f*f);
            if (c*c + d*d > 1e-3)
            {
                fftSrc[i] = ( a*c + b*d ) / ( c*c + d*d ) / multi;
```

```
                fftSrc[i+1] = ( b*c - a*d ) / ( c*c + d*d )/multi;
            }
        }

        //对结果图像进行反FFT
        fourn(fftSrc,nn,2,-1);

        //确定归一化因子
        MaxNum = 0.0;
        for (j = 0;j < lHeight ;j++)
        {
            for(i = 0;i < lLineBytes ;i++)
            {
                fftSrc[(2*lLineBytes)*j + 2*i + 1] =
                    sqrt(fftSrc[(2*lLineBytes)*j + 2*i + 1] * fftSrc[(2*lLineBytes)*j + 2*i +
1]\
                        +fftSrc[(2*lLineBytes)*j + 2*i + 2] * fftSrc[(2*lLineBytes)*j + 2*i
+ 2]);
                if( MaxNum < fftSrc[(2*lLineBytes)*j + 2*i + 1])
                    MaxNum = fftSrc[(2*lLineBytes)*j + 2*i + 1];
            }
        }

        //转换为图像
        for (j = 0;j < lHeight ;j++)
        {
            for(i = 0;i < lLineBytes ;i++)
            {
                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

                *lpSrc = (unsigned char) (fftSrc[(2*lLineBytes)*j + 2*i + 1]*255.0/MaxNum );
            }
        }

        delete fftSrc;
        delete fftKernel;
        delete fftNoise;
        // 返回
        return true;
    }
```

对应的菜单事件处理函数为：

```
void CCh1_1View::OnRestoreWiener()
{
    //图像维纳滤波复原操作

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;
```

```
	// 指向DIB像素指针
	LPSTR   lpDIBBits;

	// 锁定DIB
	lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

	// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的复原操作，其他的可以类推）
	if (::DIBNumColors(lpDIB) != 256)
	{
		// 提示用户
		MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

		// 解除锁定
		::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

		// 返回
		return;
	}

	// 更改光标形状
	BeginWaitCursor();

	// 找到DIB图像像素起始位置
	lpDIBBits = ::FindDIBBits(lpDIB);

	// 调用WienerDIB()函数对DIB进行复原
	if (WienerDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
	{

		// 设置脏标记
		pDoc->SetModifiedFlag(TRUE);

		// 更新视图
		pDoc->UpdateAllViews(NULL);
	}
	else
	{
		// 提示用户
		MessageBox("分配内存失败或图像尺寸不符合要求！", "系统提示", MB_ICONINFORMATION |
MB_OK);
	}

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();

}
```

下面是用维纳滤波进行图像复原的例子。如图 10－7、图 10－8、图 10－9、图 10－10

所示。







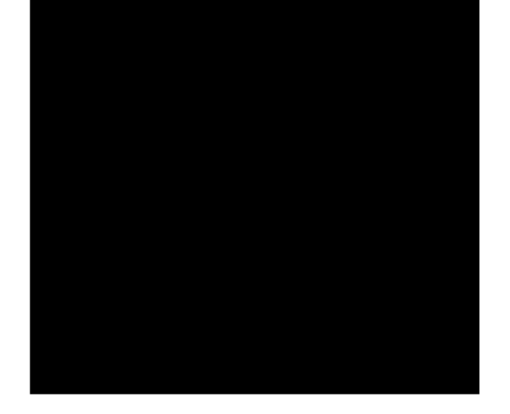


图 10—9 逆滤波处理的结果图像

图 10—10 维纳滤波处理的结果图像

10.4 非线性复原方法

前面二节介绍的还原方法有一个显著的特点是约束方程和准则函数中的表达式都可改写为矩阵乘法。这些矩阵都是分块循环阵，从而可实现对角化。下面两小节介绍的方法则都属于非线性复原方法，所采用的准则函数都不能用 W 进行对角化，因而线性代数的方法在这里是不适用的。

10.4.1 最大后验复原

设 $f(x, y)$ 和 $g(x, y)$ 都作为随机场。根据贝叶斯判决理论可知，若 $\hat{f}(x, y)$ 使下式最大：

$$\max_f p(f \mid g) = \max_f p(f \mid g)p(f)/p(g) = \max_f p(g \mid f)p(f)$$

其中 p()代表概率密度。

则 $\hat{f}(x,y)$ 可代表已知退化图像 $g(x,y)$ 时，最大后验估值意义下对原图像的估计。根据这一准则导出的滤波复原方法称为最大后验复原。

在最大后验复原中，将f、g 看为非平稳随机场。通过假设图像模型是一个平稳随机场对一个不平稳的均值作零均值 Gauss 起伏，可得出求解迭代序列：

$$\hat{f}_{k+1} = \hat{f}_k - h * \sigma_n^{-2}[g - h * \hat{f}_k] - \sigma_f^{-2}[\hat{f}_k - \bar{f}]$$

$\bar{f}(x,y)$ 是随空间而变的均值，σ_f^{-2} 和 σ_n^{-2} 分别是f 和 n 的方差的倒数，k 是叠代指数。

10.4.2 最大熵复原

前面已经指出，由于反向滤波法的病态性，复原出的图像经常具有灰度变换较大的不均匀区域。10.3 节中介绍的方法是最小化的一种反映图像不均匀性的准则函数。下面介绍的另一种方法是通过最大化某种反映图像平滑性的准则函数来作为约束条件，以解决图像复原中的病态。

首先我们假定图像函数具有非负值，即

$$f(x,y) \geq 0$$

定义一幅图像的总能量 E 为

$$E = \sum_x \sum_y f(x,y) \tag{10-7}$$

同时我们定义图像的熵为

$$H_f = -\sum_x \sum_y f(x,y) \ln f(x,y)$$

再定义噪声熵为

$$H_n = -\sum_x \sum_y n_T(x,y) \ln n_T(x,y) = -\sum_x \sum_y (n(x,y)+B) \ln(n(x,y)+B)$$

其中 B 为最小的噪声负值，以便使定义中的对数有意义(注意我们认为 0ln0＝0)。易见图像熵和噪声熵的定义非常类似于信息论中的香农熵。易知，在满足式(5．7)条件的情况下，图像熵必然在图像函数均匀分布时达到最大值。对噪声熵来说，类似的结论也是成立的。这就给我们一个提示，可以利用图像熵和噪声熵来刻划图像的平滑性或均匀性。

因此问题是如何在满足式 10－7 和图像退化模型的约束条件下使复原后的图像的图像熵和噪声熵最大。

引入如下的拉格朗日函数：

$$R = H_f + \rho H_n + \sum_{x=1}^{N}\sum_{y=1}^{N}\lambda_{mn}\left\{\sum_{x=1}^{N}\sum_{y=1}^{N}h(m-x,n-y)f(x,y) + n_T(m,n) - B - g(m,n)\right\}$$
$$+\beta\left\{\sum_{x=1}^{N}\sum_{y=1}^{N}f(x,y) - E\right\}$$

式中 $\lambda_{mn}(m, n=1, 2, .. N)$和 β是拉格朗日乘子，ρ是加权因子，用于强调 H_f和 H_n之间的相互作用关系。

使用如下的极值条件：

$$\frac{\partial R}{\partial f(x,y)} = 0$$

$$\frac{\partial R}{\partial n_T(x,y)} = 0$$

可得到与极值点$f(x, y)$、$n_T(x, y)$有关的一组方程组：

$$\tilde{f}(x,y) = \exp[-1 + \beta + \sum_{m=1}^{N}\sum_{n=1}^{N}\lambda_{mn}h(m-x,n-y)], x, y = 1,2,\ldots,N$$

$$\tilde{n}_T(x,y) = \exp(-1 + \lambda_{xy}/\rho), x, y = 1,2,\ldots,N$$

$$\sum_{m=1}^{N}\sum_{n=1}^{N}\lambda\tilde{f}(x,y) = E$$

$$\sum_{m=1}^{N}\sum_{n=1}^{N}h(m-x,n-y)\tilde{f}(x,y) + \tilde{n}_T(m,n) - B = g(m,n), m,n = 1,2,\ldots,N$$

使用迭代方法在一定的条件下总能得到上述方程组的解，从而获得复原后的图像。这种方法称为最大熵复原方法。它还有其他变化形式，例如定义不同形式的熵可获得不同的复原方法。

最大熵复原方法隐含了正值约束条件，使复原后的图像比较平滑。这种复原方法的效果比较理想，但缺点是计算量太大。

10.4.3 投影复原方法

如上节讨论的图像退化系统可用以向量表示一样，无论线性或非线性变质系统，都可以用一代数方程组来描述。

$$g(x,y) = D[f(x,y)] + n(x,y) \qquad (10-8)$$

其中$f(x, y)$是原景物图像；$g(x, y)$是变质图像；$n(x, y)$是系统噪声；D是变质算子，表示对景物进行某种运算。

图像复原的目的是解（10－8）式方程，找出f（x，y）的最好估值。

非线性代数复原方法中一个有效方法是迭代法，下面介绍的投影复原方法就是迭代法之一。

所谓迭代法是首先假设一个初始估值$f^{(0)}$（x，y），然后进行迭代运算。第k次迭代值$f^{(k)}$（x，y）仍由其前次迭代法$f^{(k+1)}$（x，y）和决定，一个最好的初始估值可能是

$$f^{(0)}(x,y)=g(x,y)$$

假设变质算子是非线性的，并忽略噪声。则（10－8）式可写成如下形式：

$$a_{11}f_1+a_{12}f_2+\dots+a_{1N}f_N=g_1$$

$$a_{21}f_1+a_{22}f_2+\dots+a_{12}f_N=g_2$$

……

$$a_{M1}f_1+a_{M2}f_2+\dots+a_{MN}f_N=g_M \qquad (10-9)$$

其中f_1和g_1分别是景物f（x，y）和退化图像g（x，y）的采样，a_{ij}为常数。由f（x，y）和g（x，y）的采样数目分别为M和N，现在需要找到f_1的最好估值，采用投影迭代法实现。

投影复原方法可以从几何学观点进行解释。$\boldsymbol{f}$=[f_1，f_2，……，f_N]可看成在N维空间中的一个向量和一点，而（10－9）式中的每一个方程式代表一个超平面，我们选取初始估值为

$\boldsymbol{f}^{(0)}$=[$f_1^{(0)}$，$f_2^{(0)}$，……，$f_N^{(0)}$]，

通常取$\boldsymbol{f}^{(0)}$=[g_1，g_2，……，g_N]，

那么下一个推测值$\boldsymbol{f}^{(1)}$取$\boldsymbol{f}^{(0)}$在第一个超平面

$$a_{11}f_1+a_{12}f_2+\dots+a_{1N}f_N=g_1$$

上的投影，即：

$$f^{(1)}=f^{(0)}-\frac{(f^{(0)}\bullet a_1-g_1)}{a_1\bullet a_1}a_1$$

其中$\boldsymbol{a}_1$=[a_{11}，a_{12}，……，a_{1N}] 以及圆点代表向量的点积，然后我们再取$\boldsymbol{f}^{(1)}$在第二超平面

$$a_{21}f_1+a_{22}f_2+\dots+a_{12}f_N=g_2$$

上的投影，并称之为$\boldsymbol{f}^{(2)}$，依次继续下去，直到得到$\boldsymbol{f}^{(M)}$满足（10－9）式中最后一个方程式，这就实现了迭代的第一个循环，然后再从（10－9）式中第一个方程式中开始第二次迭代。即取$\boldsymbol{f}^{(M)}$在第一个超平面

$$a_{11}f_1+a_{12}f_2+\dots+a_{1N}f_N=g_1$$

上的投影，并称之为$\boldsymbol{f}^{(M+1)}$，再取$\boldsymbol{f}^{(M+1)}$在

$$a_{21}f_1+a_{22}f_2+\dots+a_{12}f_N=g_2$$

上的投影，……直到（10-9）中最后一个方程式，这就实现了第二个迭代循环。接着上述方法连续不断地迭代下去，便可得一系列向量$\boldsymbol{f}^{(0)}$，$\boldsymbol{f}^{(M)}$，$\boldsymbol{f}^{(2M)}$……可以证明，对于任何给定的N，M和a_{ij}，向量$\boldsymbol{f}^{(kM)}$将收敛于$\boldsymbol{f}$，即

$$\lim_{k\to\infty} f^{(KM)} = f$$

而且，如果（10－9）式有唯一解，那么f就是这个解。如果（10－9）式有无穷多个解，那么f是使下式取最小值的解

$$\| f - f^{(0)} \|^2 = \sum_{j=1}^{N} (f_i - f_i^{(0)})^2$$

$$= (f_1 - f_1^{(0)})^2 + (f_1 - f_1^{(0)})^2 + \ldots + (f_N - f_N^{(0)})^2$$

由上可见，投影迭代法要求有一个好的初始估值$\boldsymbol{f}^{(0)}$开始迭代，才能获得好的结果。

在应用此法进行图像复原时，还可以很方便地引进一些先验信息附加的约束条件，例如$f_i \geq 0$或f_i限制在某一范围之内等。

10.4.4 Monte Carlo 复原方法

Monte Carlo复原方法是利用统计学上著名的Monte Carlo计算方法进行图像复原。这种方法的主要思想是把图像分成许多细胞，相当于像元，同时认为图像的灰度是由颗粒组成，每个颗粒具有一定的能量d_0，并假定图像的颗粒总数是已知的，即图像的总能量是已知的。某一颗粒随机地分配到某一细胞中，满足一定的判决准则。如果所有的颗粒都分配完了，那么最后的目标图像也就复原出来了。下面介绍Monte Carlo复原方法的具体计算过程。

对于一维数据情况，成象过程满足下面方程：

$$g(y_m) = f(x_n) * h(y_m - x_n) + n_0$$

其中$f(x_n)$为目标函数，由$\boldsymbol{X_n}=(x_1, x_2, \cdots\cdots x_N)$ 等N个点组成。$g(y_m)$为目标函数，由$\boldsymbol{y_m}=(y_1, y_2, \cdots\cdots y_M)$ 等M个点组成。$(y_m\text{-}x_n)$为线性移不变系统的点扩展函数，n_0为随机噪声。

假定原始图像或者称之为目标函数$f(x_n)$是由中心位于$\boldsymbol{X_n}=(x_1, x_2, \cdots\cdots x_N)$的各个细胞组成，每个细胞中包含一定数量的具有单位能量d_0的颗粒。开始时，我们认为整个目标空间是空的，假定第一个颗粒分配到细胞$\boldsymbol{X_n}$中，那么形成的模糊图像为：

$$g^{(1)}(y_m) = d_0 * h(y_m - x_n)$$

第二个颗粒分配到某一细胞$\boldsymbol{X_n}$后形成的模糊图像为：

$$g^{(2)}(y_m) = g^{(1)}(y_m) + d_0 * h(y_m - x_n)$$

同理，第k个颗粒分配到某一细胞$\boldsymbol{X_n}$中形成的模糊图像为：

$$g^{(k)}(y_m) = g^{(k-1)}(y_m) + d_0 * h(y_m - x_n) \qquad (10-10)$$

第k个颗粒是否可以分配到细胞$\boldsymbol{X_n}$中，要看是否满足一定的判决准则。这里介绍的一个

准则是：如果一个颗粒可以分配到细胞 X_n 中，那么应可以找出一个最小的因子 r，使得该颗粒分配后形成的模糊图像小于 $r \cdot g$（y_m）。在数学上可以把该准则表示为：

$$g^{(k)}(y_m) \le r \bullet g(y_m),\quad m = 1,2,\dots,M$$

如果第 k 个颗粒满足（10－10）式的判决准则，则该颗粒可以分配到细胞 X_n 中，那么形成的目标图像为

$$\hat{f}^{(k)}(x_n) = \hat{f}^{(k-1)}(x_n) + d_0$$

其中 $\hat{f}(x_n)$ 为目标的估值。

按照上述步骤，把所有颗粒分配完以后，我们也就复原出真实目标图像 $\hat{f}(x_n)$ 了。

该复原方法具有使用灵活、容易满足和实现正值约束及限带等许多约束条件且计算速度快等优点。在无噪声或噪声较小，点扩展函数和图像数据存在零点以及二值图像的情况下，可取得较好复原效果。

10.5 几种其他图像复原技术

前边讨论了几种基本的图像复原技术。除此之外，尚有一些其他的空间图像复原方法，本节将对这些方法作一些简要的讨论。

10.5.1 几何畸变校正

在图像的获取或显示过程中往往会产生几何失真。例如：成像系统有一定的几何非线性。这主要是由于视像管摄像机及阴极射线管显示器的扫描偏转系统有一定的非线性，因此会造成如图 10－11 所示的枕形失真或桶形失真。图（a）为原始图像；图（b）和图（c）为失真图像。

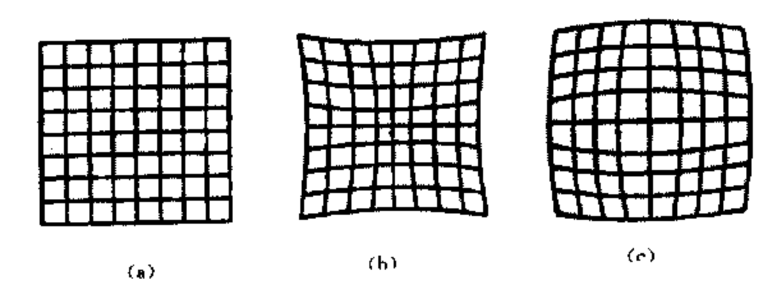


图 10－11 几何畸变

除此之外，还有由于斜视角度的原因造成获得的图像的透视失真。另外，卫星摄取的地球表面的图像往往覆盖较大的面积，由于地球表面呈球形，这样摄取的平面图像也将会有较大的几何失真。对于这些图像必须加以校正，以免影响分析精度。

由成像系统引起的几何畸变的校正有两种方法：一种是预畸变法，这种方法是采用与畸变相反的非线性扫描偏转法，用来抵消预计的图像畸变；另一种是所谓的后验校正方法。这种方法是用多项式曲线在水平和垂直方向去拟合每一畸变的网线，然后求得反变化的校正函数，用这个校正函数即可校正畸变的图像。图像的空间几何畸变及其校正过程如图 10－12 所示。

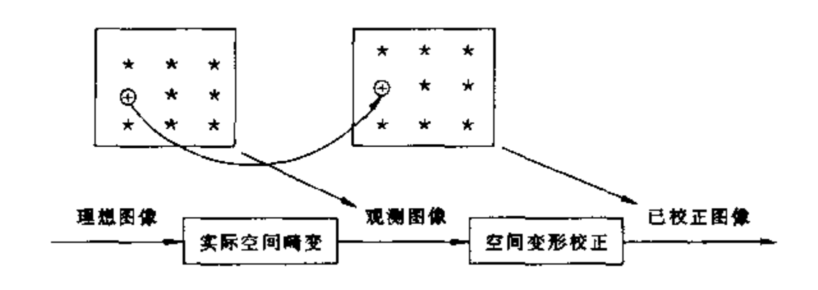


图 10－12 空间几何畸变及校正的概念

任意几何失真都可由非失真坐标系 (x, y) 变换到失真坐标系 (x', y') 的方程来定义。方程的一般形式为

$$\begin{cases} x' = h_2(x, y) \\ y' = h_2(x, y) \end{cases}$$

在透视畸变的情况下，变换是线性的，即

$$\begin{cases} x' = ax + by + c \\ y' = dx + ey + f \end{cases}$$

设 $f(x, y)$ 是无失真的原始图像，$g(x', y')$ 是 $f(x, y)$ 畸变的结果，这一失真的过程是已知的并且用函数 h_1 和 h_2 定义。于是有：

$$g(x', y') = f(x, y)$$

这说明在图像中本来应该出现在像素 (x, y) 上的灰度值由于失真实际上却出现在 (x', y') 上了。这种失真的复原问题实际上是映射变换问题。在给定了 $g(x', y')$，$h_1(x, y)$ 和 $h_2(x, y)$ 的情况下，其复原处理如下：

1. 对于 $f(x, y)$ 中的每一点 (x_0, y_0)，找出在 $g(x', y')$ 中相应的位置：

$(\alpha,\beta)=[h_1(x_0,y_0),h_2(x_0,y_0)]$。

由于α和β不一定是整数，所以通常(α,β)不会与 $g(x',y')$中的任何点重合。

2. 找出$g(x',y')$中与(α,β)最靠近的点(x_1',y_1')，并且令$f(x_0,y_0)=g(x_1',y_1')$，

   也就是把$g(x_1',y_1')$点的灰度值赋于$f(x_0,y_0)$。如此逐点进行下去，直到完成整个图像，则几何畸变得到校正。

3. 如果不采用 2 中的灰度值的代换方法也可以采用内插法。这种方法是假定(α,β)点找到后，在$g(x',y')$中找出包围着(α,β)的 4 个邻近的数字点：(x_1',y_1')，(x_{1+1}',y_1')，(x_1',y_{1+1}')，(x_{1+1}',y_{1+1}')并且有：

$$\begin{cases}x_1'\le\alpha\le x_{1+1}'\\ y_1'\le\beta\le y_{1+1}'\end{cases}$$

$f(x,y)$中点(x_0,y_0)的灰度值由$g(x',y')$中 4 个点的灰度值间的某种内插法来确定。

在以上方法的几何校正处理中，如果(α,β)处在图像$g(x',y')$之外，则不能确定其灰度值，而且校正后的图像多半不能保持其原来的矩形形状。

以上讨论的是在 g，h_1，h_2 都知道的情况下可采用的几何畸变的校正方法。如果只知道 g，而 h_1 和 h_2 都不知道，但有类似规则的网格图案可供参考利用，那么就有可能通过测量 g 中的网格点的位置来决定失真变换的近似值。

例如：如果给出了三个邻近网格点构成的小三角形，其在规则网格中的理想坐标为(r_1,s_1)、(r_2,s_2)、(r_3,s_3)，并设这些点在 g 中的位置分别为(u_1,v_1)、(u_2,v_2)、(u_3,v_3)。由线性变换关系：

$$\begin{cases}x'=ax+by+c\\ y'=dx+ey+f\end{cases}$$

可认为它把三个点映射到它们失真后的位置，由此构成如下 6 个方程：

$$\begin{cases} u_1 = ar_1 + bs_1 + c \\ v_1 = dr_1 + es_1 + f \\ u_2 = ar_2 + bs_2 + c \\ v_2 = dr_2 + es_2 + f \\ u_3 = ar_3 + bs_3 + c \\ v_3 = dr_3 + es_3 + f \end{cases}$$

解这 6 个方程可求得 a、b、c、d、e、f。这种变换可用来校正 g 中被这三点联线包围的三角形部分的失真。由此对每三个一组的网格点重复进行上述步骤，即可实现全部图像的几何校正。

10.5.2 盲目图像复原

多数的图像复原技术都是以图像退化的某种先验知识为基础的，也就是假定系统的脉冲响点扩展函数的确定方法应是已知的。但是，在许多情况下难以确定退化的点扩散函数。在这种情况下，必须从观察的图像中以某种方式抽出退化信息，从而找出图像复原方法。这种方法就是所谓的盲目图像复原。

对具有加性噪声的模糊图像作盲目图像复原的方法有两种：直接测量法和间接估计法。

直接测量法盲目图像复原通常要测量图像的模糊脉冲响应和噪声功率谱或协方差函数。在所观察的景物中，往往点光源能直接指示出冲激响应。另外，图像边缘是否陡峭也能用来推测模糊冲激响应。在背景亮度相对恒定的区域内测量图像的协方差可以估计出观测图像的噪声协方差函数。

间接估计法盲目图像复原类似于多图像平均法处理。例如：在电视系统中，观测到的第 i 帧图像为

$$g_i(x,y) = f_i(x,y) + n_i(x,y)$$

式中 $f_i(x,y)$ 是原始图像，$g_i(x,y)$ 是含有噪声的图像，$n_i(x,y)$ 是加性噪声。如果原始图像在 M 帧观测图像内保持恒定，对 M 帧观测图像求和，得到下式之关系：

$$f_i(x,y) = \frac{1}{M}\sum_{i=1}^{M} g_i(x,y) - \frac{1}{M}\sum_{i=1}^{M} n_i(x,y)$$

当 M 很大时，上式右边的噪声项的值趋向于它的数学期望值 $E\{n(x,y)\}$。一般情况下白色高斯噪声在所有 (x,y) 上的数学期望等于零。因此，合理的估计量是：

$$f_i(x,y) = \frac{1}{M}\sum_{i=1}^{M} g_i(x,y)$$

盲目图像复原的间接估计法也可以利用时间上平均的概念去掉图像中的模糊。如果有一成像系统，其中相继帧含有相对平稳的目标退化，这种退化是由于每帧有不同的线性位移不变冲激响应$h_i(x,y)$引起的。例如：大气湍流对远距离物体摄影就会产生这种图像退化。只要物体在帧间没有很大移动并每帧取短时间曝光，那么第i帧的退化图像可表示为：

$$g_i(x,y)=f_i(x,y)*h_i(x,y)$$

式中$f_i(x,y)$是原始图像，$g_i(x,y)$是退化图像，$h_i(x,y)$是点扩展函数，代表卷积。式中$i=1$，2，……，M。退化图像的傅立叶变换为

$$G_i(u,v)=F_i(u,v)*H_i(u,v)$$

利用同态处理方法把原始图像的频谱和退化传递函数分开，则可得到：

$$\ln[G_i(u,v)]=\ln[F_i(u,v)]+\ln[H_i(u,v)]$$

如果帧间退化冲激响应是不相关的，则可得到下面的和式：

$$\sum_{i=1}^{M}\ln[G_i(u,v)]=M\ln[F_i(u,v)]+\sum_{i=1}^{M}\ln[H_i(u,v)]$$

当M很大时，传递函数的对数和式接近于一恒定值，即

$$K_H(u,v)=\lim_{M\to\infty}\sum_{i=1}^{M}\ln[H_i(u,v)]$$

因此，图像的估计量为

$$\hat{F}_i(u,v)=\exp\{\frac{K_H(u,v)}{M}\}\prod_{i=1}^{M}[G_i(u,v)]^{\frac{1}{M}}$$

对上式取傅立叶反变换就可得到空域估计式$f(x,y)$。

在上面分析中，并没考虑加性噪声分量。若加以考虑，则无法进行分离处理，后边的推导也就不成立了。对于这样的问题，可以对观测到的每帧图像先进行滤波处理，去掉噪声，然后在图像没有噪声的假设下再进行上述处理。

10.6 点扩展函数的确定

10.6.1 几种典型的点扩展函数

如前所述，图像复原的一个重要特点是需要有关退化过程的先验知识。反映在滤波器设

计中，即是要求得点扩展函数(PSF)。本节介绍几种典型图像退化过程的点扩展函数。

一、均匀直线运动模糊下的 PSF

这种退化过程发生在成像设备或记录媒体和景物之间存在相对运动。不妨设成像设备或记录媒体不动，而图像 $f(x,y)$ 发生平面运动。设其在 x 和 y 方向上运动的时间变化分量为 $x_0(t)$ 和 $y_0(t)$。

假设 T 为成像设备或记录媒体的曝光时间，则退化的图像 $g(x, y)$ 应为：

$$g(x,y)=\int_0^T f[x-x_0(t),y-y_0(t)]\mathrm{d}t$$

因而可得出点扩展函数为:

$$H(u,v)=\int_0^T \exp[-j2\pi(ux_0(t)+vy_0(t))]dt$$

若图像的运动在 x 和 y 方向上是均匀直线运动，其速率为 $(v_x, v_y)=(a/T, b/T)$。则 $(x_0(t),y_0(t))=(at/T,\ bt/T)$,代入上式后有 $H(u,v)=\dfrac{T\sin\pi(au+bv)}{\pi(au+bv)}\mathrm{e}^{-\mathrm{j}\pi(au+bv)T}$

二、聚焦系统下的 PSF

由光学系统散焦造成的转移函数可用第一类一阶 Bessel 函数除以它的变量来表示。即：

$$H(u,v)=J_1(\pi\ d\rho)/\pi\ d\rho$$

式中 $\rho=\sqrt{u^2+v^2}$。光学系统散焦的 PSF 在线性位移不变系统中是圆函数，d 是圆函数的直径，$J_1(*)$ 是第一类一阶 Bessel 函数。

长时间曝光下由于大气湍流可引起成像模糊。这一退化过程的点扩展函数为:

$$H(u,v)=\exp[-c(u^2+v^2)^{5/6}]$$

c 是与湍流性质有关的常数。

10.6.2 系统辨识

在进行图像复原以前，要求模糊函数的 PSF 已知。PSF 在某些情形下是已知的，在上一节中介绍了一些典型的点扩展函数。但在别的情况下则需要根据退化图像通过实验来确定。在本节中，我们将介绍一些确定一个成像系统的 PSF 和 MTF 的方法。

一、通过测试靶进行系统辨识

在许多情况下，系统的传递函数可以在系统投入使用之前直接测定。假设对图 10－13 的系统说来，冲激响应 $h(x, y)$ 为未知且需要测定。如果有一个合适的测试信号 $f(x, y)$，就可用下式直接找出传递函数：

$$H(u,v)=\frac{G(u,v)}{F(u,v)}$$

最理想的是，$F(u,v)$ 没有零点。如果它有零点，但 $H(u,v)$ 相对光滑，我们仍然可用数值方法解方程。

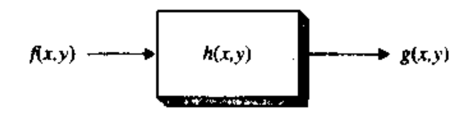


图 10－13 一个线性系统

- 点源测试靶

如果我们能输入一个冲激（点源），则系统的输出就是冲激响应（即 PSF）。尽管冲激在物理上是不可能实现的，但我们可以用一个与 PSF 本身相比足够窄的脉冲来代替。例如：星体可用于望远镜的 PSF 的测量，只是由于大气造成的模糊会使 PSF 比其本来显得更大。我们可以用一个亮的 LED 点光源来进行照相机透镜系统的测定，而荧光小珠子则可用于显微镜的辨识。但是，一般来说采用点光源靶进行光学系统 PSF 的直接测量——特别对于亮场显微镜——是行不通的，因而必须采用其他方法。

- 正弦波测试靶

确定传递函数最可靠的方法之一是使用正弦型输入函数。设输入为：

$$f(x,y)=\cos(2\pi s_0 x)$$

它是具有正弦轮廓的竖直条状图案。由于此输入在 y 方向恒定，则输出为：

$$g(x,y)=H(s_0,0)\cos(2\pi s_0 x)$$

输出的频谱为

$$G(u,v)=H(s_0,0)[\delta(u-s_0)+\delta(u+s_0)]\delta(v)$$

这是一个位于 u 轴上 $u=\pm S_0$ 处的偶冲激对。

通过在不同频率和不同朝向的情况下重复此过程，可得到任意精度要求的传递函数。此外，对于圆对称的或可分离的传递函数的情况，工作量可大大减少。实际上，可通过输入一个含几个不同频率的水平和垂直正弦条图案来完成全部工作。这样的一幅输入图像叫作正弦波（测试）靶。由于产生这样的靶较困难，特别是用于测量显微镜的 PSF 时，要求尺寸较小，有时也用条状靶来测出系统的方波响应，再用它来近似系统的传递函数。

- 线测试靶

设系统输入为一沿 y 轴的无限窄直线。此时输入可表示为：

$$f(x,y)=\delta(x)$$

它可被认为是 x 方向 δ 函数与 y 方向常数之积。于是输出由以下卷积给出

$$g(x,y)=\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}h(p,q)\delta(x-p)dpdq=\int_{-\infty}^{\infty}h(x,y)dy$$

而输出频谱为

$$G(u,v)=H(u,v)\delta(v)=H(u,0)\delta(v)$$

因此，线输入函数产生的输出等于系统冲激响应对其 y 分量的积分。根据二维傅立叶变换的投影性质，输出的频谱仅是沿 u 轴计算的传递函数。

如果 $h(x, y)$ 是圆对称的，则传递函数 $H(u, v)$ 可完全由任意方向上的一个线输入所产生的线扩散函数来确定。如果 $h(x, y)$ 可分离为 x 方向和 y 方向函数的乘积，则系统的垂直和水平线扩散函数足以确定系统的传递函数。

如果 $h(x, y)$ 是不对称的，二维傅立叶变换的旋转特性使我们可以沿各个转角计算线扩散函数，对其变换以获得此角度上 $H(u, v)$ 的轮廓，再进一步重建传输函数。这一技术构成了将在第 11 章讨论的计算机断层重建技术的基础。

● 边缘测试靶

设输入幅度包含一个沿 y 轴的由低到高的突变。这个输入可表示为在 x 方向的阶跃函数与 y 方向的常数之积，可写为：

$$f(x,y)=u(x)$$

其中 $u(x)$ 为引入的阶跃函数。由于边缘函数为线函数的积分，两卷积与积分及求导可互换，因此边缘扩散函数为线扩散函数的积分。这样，就可以对边缘扩散函数求导再按上节方法处理。另一种作法是，利用在付立叶变换中积分相当于引入一个 $1/j2\pi s$ 的性质，得到

$$G(u,v)=\frac{H(u,0\delta(v)}{j2\pi u}$$

由它可得到 u 不为零处的传递函数。

● 频率扫描测试靶

另一个与正弦波靶类似，不需要对输出进行变换来计算传递函数的输入是频率扫描测试靶。为便于说明，考察图 10－14 中的一维线性系统。

输入一个频率随与原点的距离线性增加的调和信号。一个频率为 ax 的调和信号用下式表示：

$$f(x)=e^{j2\pi ax^2}$$

输出信号由卷积给出：

$$g(x)=\int_{-\infty}^{\infty}h(\tau)e^{j2\pi a(x-\tau)^2}d\tau=e^{j2\pi ax^2}\int_{-\infty}^{\infty}h(\tau)e^{j2\pi a\tau^2}e^{-j4\pi ax\tau}d\tau$$

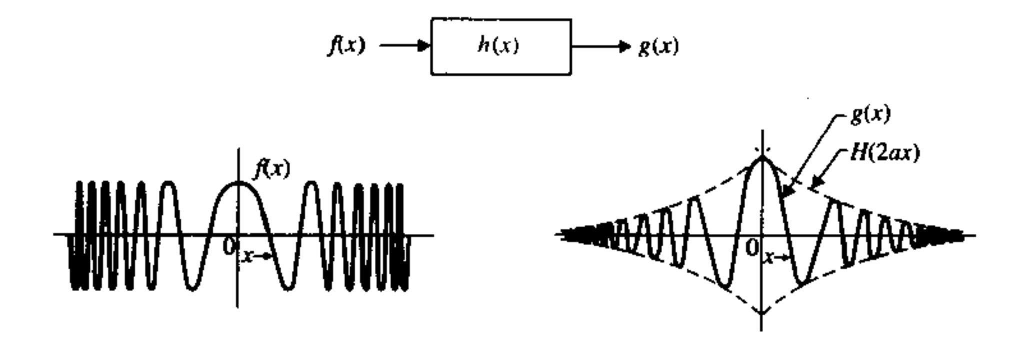


图 10－14 频率扫描输入的系统辨识

这里第二个表达形式是由展开指数中的平方项得到的。作如下替换：

$s = 2a\tau$

$ds = 2ad\tau$

$\tau = \dfrac{s}{2a}$

$d\tau = \dfrac{ds}{2a}$

并注意到积分号前的项其实就是输入，则有：

$$g(x) = \frac{1}{2a} f(x) \int_{-\infty}^{\infty} h[\frac{s}{2a}] e^{\frac{j\pi s^2}{2a}} e^{-j2\pi sx} ds$$

上式中的积分项实际上为一个乘积函数的傅立叶变换，它可写为：

$$g(x) = \frac{1}{2a} f(x) \Im\{h[\frac{s}{2a}] e^{\frac{j\pi s^2}{2a}}\} \tag{10-11}$$

如果冲激响应在区间（$-T$，T）以外趋于零，则

$$h[\frac{s}{2a}] \approx 0 \quad |s| > 2aT$$

进一步，若

$$\frac{(2aT)^2}{2a} = 2aT^2 << 1 \tag{10-12}$$

则有

$$e^{\frac{j\pi s^2}{2a}} \approx 1 \quad |s| \le 2aT$$

于是输出可简化为

$$g(x) = \frac{1}{2a} f(x) 2aH(2ax) = f(x)H(2ax)$$

即仅是受到传递函数包络的输入（即输入与传输函数的乘积）。

式（10－12）所作的假设可以从两方面来解释。首先，它意味着冲激响应与频率扫描的第一个周期相比是较窄的。根据相似性原理，这等于假设传递函数与频率扫描的第一个周期相比很宽。如果第二个条件不成立，就难于观察到输出的包络 $H(2ax)$。

注意，在式（10－12）假设下采用频率扫描靶使我们不必计算傅立叶变换就可确定传递函数，另一方面，如果我们愿意进行傅立叶变换的话，我们就可不必作此假设。回到式（10－11），如果在等式两端同除以 $f(x)$，再进行傅立叶变换，就得到

$$\Im^{-1}\{\frac{g(x)}{f(x)}\}=\frac{1}{2a}h[\frac{s}{2a}]e^{\frac{j\pi s^2}{2a}}$$

如对上式的两边取幅值，则复指数项消失，得到

$$|\Im^{-1}\{\frac{g(x)}{f(x)}\}|=\frac{1}{2a}h[\frac{s}{2a}]$$

由此很容易地解出冲激响应 $h(x)$ 的值。这看来有些奇怪：将输出与输入之比进行变换得到的是冲激响应而非传递函数。

二、用互相关进行系统辨识

假定像图 10－15 那样，我们将一个线性系统的输出和输入进行互相关。互相关器的输出的谱函数为：

$$Z(s)=G(s)F^*(s)=H(s)F(s)F^*(s)=H(s)P_f(s)$$

其中 $P_f(s)$ 为输入信号的功率谱。若 $f(x)$ 为非相关白噪声，则 $P_f(s)$ 为一常数，因而互相关器输出的不过是系统的冲激响应。这样，我们就可以用随机噪声图像作为系统输入，算出它与系统输出的互相关，以得到系统的 PSF。而且，互相关输出的谱函数就是系统传递函数。

可用如下方法产生一幅白噪声图像：首先产生一个具有恒定幅度和随机相位的二维谱函数，然后计算其付立叶逆变换以得到白噪声图像。

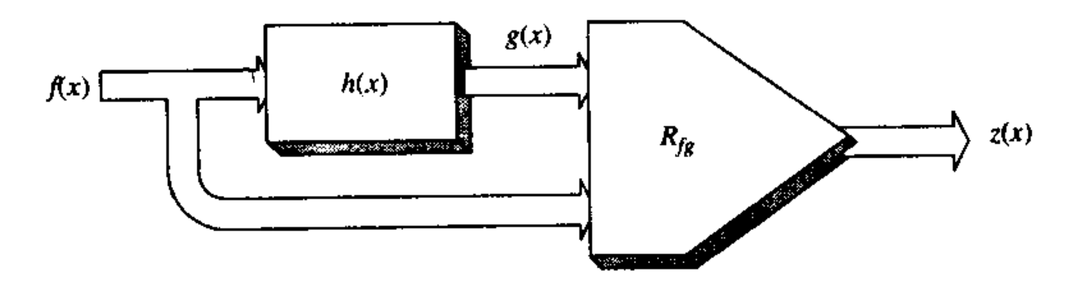


图 10－15 用互相关进行系统辨识

三、图像辨识系统

在某些情况下，要想在和当初记录某幅退化图像时的相同条件下标定成像系统是不现实甚至是不可能的。例如：对运动模糊或因大气扰动造成的随机退化；又如需要复原一张照片，而原来的相机已找不到了等。在这些情况下，只能试图根据退化的图像本身来确定退化PSF。

如果图像中包含某些可用模型进行解析描述的特征，那么在理论上可以通过对模型的去卷积得到PSF。

- 点源

如果可以设法使退化图像中包含一个点光源（或在白背景下的一个几乎看不见的小斑），就可以直接得到PSF。如果这个点光源或黑斑的尺寸不可忽视（不适宜看作冲激函数）则可考虑用高斯函数、平顶圆脉冲或其他通过去卷积能得到PSF的合适函数来描述。

这项技术针对由于扰动而严重退化的天文照片具有很大的应用价值。在这里，源（星体）是现成的，而退化程度严重足以使去卷积大有帮助。这项技术在高质量图像中有很大作用。例如：用衍射有限的照相机得到的照片中，就难以找到满足以下条件的点光源或斑点：它既足够小到显得是一个点（可以应用PSF），又仍足够大（在通过系统后仍有充足的能量保证测量精度）。它在图像中只占据一个很小的区域，特别容易受到系统噪声的干扰。因此，用图像中的点来直接测量PSF用途有限。

- 线

根据二维傅立叶变换的投影原理，线扩散函数的傅立叶变换给出一个二维传递函数的一维分量。线扩散函数的优点在于图像中的线源可沿其走向平均，因此能得出噪声相对比较低的估计。但对于高质量的系统而言，图像中的线状物体必须极端的细。因此，与其背景相比它必须非常亮（或暗）以便在通过系统后仍具有足够的幅度。

- 边缘

大部分景物图像中都含有可被视为理想边缘的特征。这样的边缘可沿其走向平均，以得出系统在某一特定方向上的、噪声相对较低的估计边扩散函数。将得到的边扩散函数进行微分可得到线扩散函数，后者通过变换可给出传递函数的一个分量。

如果已知PSF为圆对称的，则由线扩散函数得到的一维PSF可通过简单的旋转得到二维PSF。如果PSF和对应的传递函数可分解为函数的积，则一个水平边缘和一个垂直边缘就足以决定传递函数了。然而，在一般情况下，需要多个不同方向的边缘才足以确定传递函数。

由于微分是一个高通滤波器类型的运算，在得到的线扩散函数中边扩散函数的噪声会显得被放大了。因此应该尽可能沿着边缘进行平均以降低噪声影响。然而，如果边缘不完全呈直线或不平行于采样光栅，平均就会使边缘模糊，并使得传递函数具有比实际上更低通的滤波特征。

通过旋转几何变换，可使斜线与采样光栅平行。然而，除非对景物进行了足够的过采样，否则旋转中包含的内播也会使图像变得模糊。

边扩散函数的另一问题是如仅在边缘附近很窄的一个区域内处理会引起截断误差。有人指出对于一个衍射极限系统，边扩散函数必须在宽度为爱里（Airy）斑直径10倍以上的区域上计算才能使传递函数误差降到2%以下。

对边扩散函数的正确应用并不像看起来那样简单，只有通过十分小心的处理才能精确地确定传递函数。一般来说，任何从退化图像中确定PSF的处理都要谨慎进行。

四、由退化图像频谱确定 OTF

一般说来，复杂景物的图像具有相对比较平滑的幅度频谱。如果引起退化的传递函数具有零点（例如：在有线性运动模糊），这些零点就会迫使退化图像的谱在某些特定的频率上变成零。如果给出模糊函数的适当模型，则这些零值（或接近零值）在空间频率平面内的位置使我们可以确定模糊 OTF 的未知参数。有时通过采用高通滤波器的预处理方法来使谱函数零值可视化。

通过对退化图像的功率谱取对数，我们可增加由退化传递函数中零点引起的下凹的幅度。如果零点是等间距分布的，则在功率谱的对数图上，会产生一系列周期性尖峰。对数功率谱的功率谱，有时也叫作倒频谱 ceptrum，可用来确定这些尖峰的确切间距，进而得到退化传递函数的零点。

另外一种方法是首先将退化图像划分为与退化函数 PSF 相比较足够大的正方形区域，并在每个这样的区域上计算对数功率谱的均值。对于复杂景物，各信号分量会在对数功率谱的平均过程中被平均掉，而退化传递函数则不然（其在整个图像上不变）。这样，对数功率谱的平均就近似地收敛到退化传递函数幅度平方的对数。

10.7 图像系统中的噪声模型

10.7.1 噪声模型

噪声可以理解为“妨碍人们感觉器官对所接收的信源信息理解的因素”。例如一幅黑白图片，其平面亮度分布假定为 $f(x, y)$，那么对其接收起干扰作用的亮度分布 $R(x, y)$ 即可称为图像噪声。活动的黑白电视图像噪声可以表示为 $R(x, y, t)$，彩色电视图像噪声可以表示为 $R(x, y, t, \lambda)$。但是，噪声在理论上定义为“不可预测的、只能用概率统计方法来认识的随机误差”。因此将图像噪声看成是多维随机过程是合适的，描述噪声的方法完全可以借用随机过程的描述，即用其概率分布函数和概率密度分布函数进行描述。但在很多情况下，这样的描述方法很复杂，甚至是不可能的，所以在实际应用中往往也是避免使用的。通常是使用的其数字特征，即均值方差，相关函数等，因为这些数字特征都可以从某些方面反映出噪声的特征。例如：均方值 $E[R^2(\cdot)]$ 描述噪声总功率，方差 $E[R^2(\cdot)-E[R^2(\cdot)]]$ 描述噪声的交流功率，而均值的平方 $(E[R^2(\cdot)])^2$ 表示了噪声的直流功率。

在目前大多数数字图像系统中，输入光图像都是采用先冻结再扫描方式将多维图像变成一维电信号，再对其进行处理、存贮、传输等加工变换方法来进行的，最后还要再组成多维图像信号，而图像噪声也将同样受到这样的分解和合成。在这些过程中电气系统和外界影响将使得图像噪声的精确分析变得十分复杂。另一方面图像只是传输视觉信息的媒介，对图像信息的认识理解是由人的视觉系统所决定的。不同的图像噪声，人的感觉（理解）程度是不同的，这就是人的噪声视觉特性课题。这方面虽早已进行研究，但终因人的视觉系统本身未搞清楚而未获得解决。所以现在还不能规定出确切的图像噪声干扰的客观指标，而只能进行一些主观评价研究。尽管如此，图像噪声在数字图像处理技术中的重要性还是愈加明显，如高放大倍数航片的判读及 X 射线图像系统中的噪声去除等都已成为不可缺少的技术步骤。再

如在图像系统的空间频率特性等某些性能测试，图像信息的伪装以及全息技术中都有一定的应用。本节只概略介绍一下有关图像噪声的基本知识。

图像噪声按其产生的原因可分为：

1. 外部噪声：是指系统外部干扰以电磁波的形式或经电源串进系统内部而引起的噪声。如电气设备，天体放电现象等引起的噪声。
2. 内部噪声：一般可分为下列 4 种：

(1) 由光和电的基本性质所引起的噪声。如电流的产生是由电子或空穴粒子的集合所形成的。因这些粒子运动的随机性而形成的散粒噪声；导体中自由电子运动所形成的热噪声；根据光的粒子性，图像是由光量子所传输，而光量子密度随时间和空间变化所形成的光量子噪声等。

(2) 电器的机械运动产生的噪声。如各种接头因抖动引起电流变化所产生的噪声；磁头，磁带等抖动引起的抖动噪声等。

(3) 元器件材料本身引起的噪声。如正片和负片的表面颗粒性和磁带磁盘表面缺陷所产生的噪声。随着材料科学的发展，这些噪声可望不断减少，但在目前来讲，还是不可避免而要注意的。

(4) 系统内部设备电路所引起的噪声。如电源引入的交流声；偏转系统和精位电路引起的噪声等。

图像噪声从统计理论观点可分为平稳噪声和非平稳噪声两种。这两种噪声可以理解为：统计特性不随时间变化而变化的噪声称为平稳噪声。统计特性随时间变化而变化的称为非平稳噪声。还可以按噪声幅度分布形状来定义，如幅度分布是按高斯分布的就称为高斯噪声，而按雷利分布的就称为雷利噪声。当然也有按噪声频谱形状来命名的。如频谱均匀分布的噪声称为白噪声；频谱与频率成反比的称为 $1/f$ 噪声；而与频率平方成正比的称其为三角噪声等等。

另外，按噪声和信号之间关系可分为加性噪声和乘性噪声：假定信号为 $S(t)$，噪声为 $n(t)$，如果混合迭加波形是 $S(t)+n(t)$ 形式，则称此类噪声为加性噪声；如果迭加波形为 $S(t)[1+n(t)]$形式，则称为乘性噪声。前者如放大器噪声等，每一个像素的噪声不管输入信号大小，噪声总是分别加到信号上。后者如光量子噪声、胶片颗粒噪声等，由于载送每一个像素信息的载体的变化而产生的噪声受信息本身调制。在某些情况下，信号变化很小，噪声也不大，为了分析处理方便，往往将乘性噪声近似认为是加性噪声，而且总是假定信号和噪声是互相统计独立。

图像系统中噪声来自多方面，有电子元器件。如电阻引起的热噪声；真空器件引起的散粒噪声和闪烁噪声；面结型晶体管产生的颗粒噪声和 $1/f$ 噪声；场效应管的沟道热噪声；光电管的光量子噪声和电子起伏噪声；摄像管引起的各种噪声等等。由这些元器件组成各种电子线路以及构成的设备又将使这些噪声产生不同的变换而形成局部线路和设备噪声。另外还有就是光学现象所产生的图像光学噪声。我们仅就一些专用元器件和设备噪声略加介绍，其他一些噪声可在一般的电子技术文献中获得了解。

3. 光电管的噪声

光电管通常作为光学图像和电子信号之间转换器件，如光密度计各种形式的扫描输入输出设备，传真机的收发片机光电转换等。

光电管的噪声主要包括两个方面：其一是到达光电管阴极光量子数的起伏骚动，其二是每个入射光量子所发射电子数的起伏骚动。假定光电管的阳极电流为 I，据肖特基公式，阳极电流的噪声电流可由下式表示：

$$\bar{i}_n^{\,2} = 2eIdf$$

式中 e 为电子电荷，f 为频率

4. 摄像管的噪声

摄像管大体可分为三类。其一是利用光电子放电效应进行光电变换，如光电析象管、正析摄像管、超正析摄像管等，除一些特殊场合（如低照度医疗电视等）外均已很少使用。其二是利用光导效应进行光电变换。如视象管、光导摄像管、卡尼康管、硅靶管等。因为具有轻巧价廉等优点。目前广泛应用在工业电视，广播电视方面。其三是固体摄像器件。如 BBD（Bucket Brigade Device）和 CCD（Chafge Coupled Devices）。它是将光学信号电荷存储于金属氧化物电容的半导体耗尽层上，由外部加激励脉冲，使电荷沿同一方向顺序传输，从输出端取出信号电流。目前国内外均有一些试制产品在应用，就其噪声性能来讲光导摄像管最好。固体摄像器件尚需进一步改进，下面列出三类摄像管中典型输出信噪比。

(1) 超正析摄像管的输出信噪比为：

$$\frac{S}{N} = \frac{1}{\beta}\sqrt{\frac{I_p}{2eB}} \bullet \sqrt{\frac{T(\delta-1)}{\delta + \frac{1}{\delta-1} + \frac{1}{m}}}$$

式中，B 为频带宽度、T 为靶网透过率、β 为二次电子放大倍数、δ 为靶面二次电子发射比、e 为电子电荷、m 为射束调制度、I_p 为光电流。

由于 β 可能有 1000 左右的数值，所以信噪比提高就比较困难了。

(2) 光导摄像管输出信噪比为：

$$\frac{S}{N} = \frac{I_S}{\sqrt{\frac{4kTB}{R}(1 + \frac{4}{3}\pi^2 RR_e C^2 B^2}}$$

式中，k 为波尔兹曼常数、R 为输出电阻、T 为绝对温度、Re 为前级管子的等效噪声电阻、B 为频带宽度、C 为与 R 并联的寄生电容、Is 为光电流。

由于光导摄像管没有信号电流 Is 的上限，所以增加 Is 即在光线充足的场合下可以获得较好的信噪比性能。

(3) CCD 电荷耦合器件摄像管输出信噪比：

$$\frac{S}{N} = \frac{Q_S}{2\pi\, kTC}$$

式中：Qs 为一个单元所存贮的电荷量，$Qs=eNs$，Ns 为一个单元的载流子数目；C 为输入电容。

另外，由电荷传输和电荷陷阱所引起的起伏噪声，对 CCD 摄像管也有很大影响，从国

内现有的CCD摄像机来看，其噪声性能尚需大大改善。

5. 摄像机的噪声

摄像机噪声主要包括两个方面：一是摄像管输出噪声；另一部分是摄像机中放大和处理电路引起的噪声。光导管摄录管输出电流小（小于1微安），而信噪比较高，整机信噪比主要取决于放大和处理电路。超正析管摄像机，输出信号电流大（几十微安），信噪比往往低于40分贝，故放大器噪声不是主要问题。除此以外还有摄像管内部网状电极和扫描射束角度不正而产生的差拍噪声；射束冲击不良引起的闪烁噪声；由偏转和高压电路经静电感应和耦合以及地电流窜入图像信号中的同步脉冲噪声（画面上呈竖直纹状噪声）等。为了减少这种噪声，要考虑摄像管与前置放大器的屏蔽和地电流来安装配线，特别是前置放大器应尽可能接近摄像管安装，摄像管信号引出线尽可能短，外部噪声影响主要来自广播感应干扰，强电场场合下使用必须有良好屏蔽效果的摄像机外壳。彩色摄像机输出图像中的噪声取决于红、绿、蓝三通道噪声均方电压，既有亮度噪声也有色度噪声，但亮度噪声、色度噪声较小，而且它与彩色制式有关。

对摄像机输出噪声影响最大的是前置放大器的噪声性能，至于其他放大和处理电子电路中的噪声如箝位噪声、外形校正电路的噪声等，对光导摄像机影响不大。

6. 光学噪声

对于图像系统而言，光学噪声之所以重要，主要是因为在全部系统噪声中光学噪声占相当的比重。所谓光学噪声系指由光学现象产生的噪声。如胶片的粒状结构产生的颗粒噪声；印象纸粗糙表面凹凸不平所产生的亮度浓淡分布、投影屏和荧光屏的粒状结构引起的颗粒噪声等。

光学噪声和电气噪声的主要差别表现为：前者是在二维空间中展开的图形，而后者可由电压的时间变化来表示。另外光学噪声多半是乘性噪声即随信号大小而变化，而电气噪声一般可以认为是加性噪声。但两者都可以看作是平稳随机过程，故可用付立叶变换进行分析处理。

下面仅扼要分析胶片颗粒噪声：

胶片是应用广泛的图像记录介质，如卫片，航片，云图等。胶片成象是借助于卤化银的光化学变化特性、当胶片上卤化银涂层曝光后产生潜影（也称显影中心），将此胶片放入显影液中，显影中心就会沉淀为银粒子，由于银粒子沉淀的内在随机性，如银粒沉淀时尺寸和形状的随机性；银粒之间距离分布随机性等使其形成的颗粒噪声是一个随机过程，而且一般可以认为是白色噪声。假定一块小区域的平均光学密度为μ_D，其噪声标准偏差为σ_D，则光学密度d关于μ_D的高斯概率密度函数为：

$$p(d)=\frac{1}{\sigma_D\sqrt{2\pi}}\exp[-\frac{1}{2}(\frac{d-\mu_0{}^2}{\sigma_D})]$$

式中σ_D不是一个常数，而是随平均密度值的变化而变化：

$$\sigma_D=k(\mu_0)$$

这里k为系数，产值根据我们对柯达胶片测试结果为1/2－1/3。如方光阑，柯达2474胶片，产值为0.445，而柯达4421胶片的产值为0.386。

胶片的颗粒噪声与其上面的粒子结构的粒子大小、形状、浓度分布都有关系，在实际测试或观察时与其照明条件也有关。作为使用胶片者来讲，只能在胶片显影和照明光度上做些工作。如进行微粒显影可使银粒子的尺寸减小，而激活显影则可增大银粒子，从而引起噪声变大等。如果将胶片再进行电视摄像输入，除颗粒噪声外还要加上光量子噪声。这种噪声一般随照度减弱而增强，在X射线等范围内，由于射线对人体影响而不能太强。因此引入噪声就严重。

如果在摄像机上加光学透镜等光学变换系统还将产生由光栅效应引起的一些光学系统噪声。图像系统光学噪声改善一是用光学方法，如进行光学最佳设计，光学方法的空间滤波等。这在许多光学图像处理文献中均有论述。另一种是应用计算机以数字方法进行处理。

下面我们来看一下图像系统的噪声特点。

7. 噪声的扫描变换

现在图像系统的输入光电变换都是先将二维图像信号扫描变换成一维电信号再进行处理加工，最后再将一维电信号变成二维图像信号。噪声当然存在着同样的变换方式，如图10－16所示。

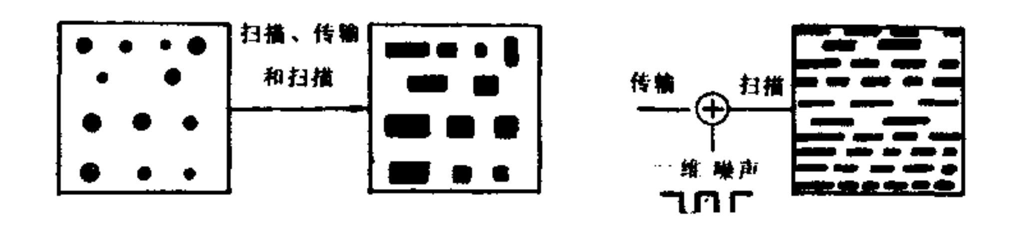


图10－16 噪声的扫描变换

噪声的水平方向和垂直方向的频率分布可由下式给定：

$$\phi_x(\omega)=\frac{1}{L}\phi_0(\frac{\omega}{L})$$

式中，ω 为空间角频率，$\phi_0(\omega)$为水平方向噪声的功率密度分布，$\phi_X(\omega)$为垂直方向噪声的功率密度分布，L为以扫描线间隔为单位的扫描线点。

由此式可见，从空间频率分布来看，垂直方向的噪声带宽为水平方向噪声带宽的L倍，因此用一般电气低通滤波器来限制噪声在垂直方向上的效果就不如水平方向的效果好。

在许多电视摄像中往往采用高频补偿（加重）技术，因为电子线路噪声多呈三角噪声，随频率增高而增加，这样使高频端噪声更为严重。

8. 噪声与图像的相关性

使用光导摄像管的摄像机，可以认为信号幅度和噪声幅度无关。而使用超正析摄像机的信号和噪声相关，黑暗部份噪声大，明亮部份噪声小。在数字图像处理技术中量化噪声是肯定存在的，它与图像相位相关，如图像内容接近平坦时，量化噪声呈现伪轮廓，但此时图像信号中的随机噪声会因颤噪效应反而使量化噪声变得不那么明显。

9. 噪声的迭加性

在串联图像传输系统中，各部份窜入噪声若是同类噪声可以进行功率相加，因此信噪比要下降。若不是同类噪声应区别对待，而且要考虑视觉检出特性的影响。但因视觉检出特性中许多问题未研究清楚，也只能进行一些主观评价试验。如空间频率特性不同的噪声迭加要考虑视觉空间频谱的带通特性。而时间特性不同的噪声迭加要考虑视觉滞留和其闪烁特性，亮度和色度噪声迭加一定要搞清楚视觉的彩色特性。这些都因为视觉特性未获解决而无法分析。

10.7.2 Visual C++编程实现

下面我们利用 Visual C++编程做一个在图像中加入噪声的实验，看一看噪声对图像质量的影响。

我们分别在图像中加入随机噪声和椒盐噪声这两种不同的噪声。菜单如图 10－17 所示。



图 10－17 噪声处理菜单

随机噪声由函数 RandomNoiseDIB()实现：

```
/*************************************************************************
 *
 * 函数名称:
 *   RandomNoiseDIB()
 *
 * 参数:
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
 * 返回值:
 *   BOOL               - 加噪声操作成功返回TRUE，否则返回FALSE。
 *
 * 说明:
 *   该函数用来对DIB图像进行加噪声操作。
 *
 ************************************************************************/

BOOL WINAPI RandomNoiseDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    // 图像每行的字节数
    LONG lLineBytes;
```

```
    //像素值
    unsigned char pixel;

    //噪声
    BYTE NoisePoint;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    //生成伪随机种子
    srand((unsigned)time(NULL));

    //在图像中加噪
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            NoisePoint=rand()/1024;

            // 指向原图像倒数第j行，第i个像素的指针
            lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

            //取得像素值
            pixel = (unsigned char)*lpSrc;

            *lpSrc = (unsigned char)(pixel*224/256 + NoisePoint);
        }
    }
    // 返回
    return true;
}
```

相应的菜单事件处理函数如下：

```
void CCh1_1View::OnRestoreRandomnoise()
{
    //图像加噪操作，在图像中加入随机噪声

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的模糊操作，其他的可以类推）
    if (::DIBNumColors(lpDIB) != 256)
```

```
    {
        // 提示用户
        MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 调用RandomNoiseDIB()函数对DIB进行加噪处理
    if (RandomNoiseDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
    {

        // 设置脏标记
        pDoc->SetModifiedFlag(TRUE);

        // 更新视图
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("分配内存失败！", "系统提示", MB_ICONINFORMATION | MB_OK);
    }

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

椒盐噪声由函数SaltNoiseDIB()实现：

```
/*************************************************************************
 *
 * 函数名称：
 *   SaltNoiseDIB()
 *
 * 参数：
 *   LPSTR lpDIBBits    - 指向原DIB图像指针
 *   LONG  lWidth       - 原图像宽度（像素数，必须是4的倍数）
 *   LONG  lHeight      - 原图像高度（像素数）
 *
```

```
* 返回值:
*   BOOL                - 加噪声操作成功返回TRUE，否则返回FALSE。
*
* 说明:
*   该函数用来对DIB图像进行加噪声操作。
*
************************************************************************/

BOOL WINAPI SaltNoiseDIB (LPSTR lpDIBBits, LONG lWidth, LONG lHeight)
{
    // 指向原图像的指针
    LPSTR   lpSrc;

    //循环变量
    long i;
    long j;

    // 图像每行的字节数
    LONG lLineBytes;

    // 计算图像每行的字节数
    lLineBytes = WIDTHBYTES(lWidth * 8);

    //生成伪随机种子
    srand((unsigned)time(NULL));

    //在图像中加噪
    for (j = 0;j < lHeight ;j++)
    {
        for(i = 0;i < lLineBytes ;i++)
        {
            if(rand()>31500)
            {
                // 指向原图像倒数第j行，第i个像素的指针
                lpSrc = (char *)lpDIBBits + lLineBytes * j + i;

                //图像中当前点置为黑
                *lpSrc = 0;
            }
        }
    }
    // 返回
    return true;
}

#undef SWAP
```

相应的菜单事件处理函数为：

```
void CCh1_1View::OnRestoreSaltnoise()
{
    //图像加噪操作，在图像中加入椒盐噪声
```

```
// 获取文档
CCh1_1Doc* pDoc = GetDocument();

// 指向DIB的指针
LPSTR    lpDIB;

// 指向DIB像素指针
LPSTR    lpDIBBits;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的模糊操作，其他的可以类推）
if (::DIBNumColors(lpDIB) != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图的运算！", "系统提示", MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 更改光标形状
BeginWaitCursor();

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 调用SaltNoiseDIB()函数对DIB进行加噪处理
if (SaltNoiseDIB(lpDIBBits, ::DIBWidth(lpDIB), ::DIBHeight(lpDIB)))
{

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 更新视图
    pDoc->UpdateAllViews(NULL);
}
else
{
    // 提示用户
    MessageBox("分配内存失败！", "系统提示" , MB_ICONINFORMATION | MB_OK);
}

// 解除锁定
::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

// 恢复光标
```

```
    EndWaitCursor();
}
```

如图 10－18、图 10－19、图 10－20 所示是一个用该程序对图像进行噪声处理的结果：



图 10－18 原始图像



图 10－19 随机噪声



图 10－20 椒盐噪声

第十一章　图像的压缩编码

在介绍图像的压缩编码之前，首先介绍一下图像压缩的必要性。众所周知，图像信息的数据量是相当庞大的。例如，地球资源卫星的一张图像就要2340×3240×7×4（4帧）≈≈212Mbit。卫星每天要获取很多这样的图片，如果不压缩直接传输到地球将是非常费时的。又如，一张A4(210mm×297mm)幅面的图片，若用中等分辨率（300dpi）的扫描仪按真彩扫描，其数据量为：（300×210÷25.4）×（300×297÷25.4）×3≈26MB，这也不是一个小数目。

大数据量的图像信息会给存储器的存储容量、通信干线信道的带宽，以及计算机的处理速度增加极大的压力。单纯靠增加存储器容量，提高信道带宽以及计算机的处理速度等方法来解决这个问题是跟不上需求的，因此就需要对图像进行压缩处理。图像数据压缩的可能性是因为图像中像素之间，行或帧之间都存在着较强的相关性。从统计观点出发，就是某个像素的灰度值（颜色）总是和其周围其他像素的灰度值（颜色）存在某种关系，应用某种编码方法提取并减少这些相关特性，这样就可以实现图像压缩。从信息论的角度来看，压缩就是去掉信息中的冗余。即，保留不确定的信息，去掉确定的信息（可推知的），也就是用一种更接近信息本质的描述来代替原有冗余的描述。

图像数据压缩的目的是节省图像存储空间、减少传输信道的容量、缩短图像加工处理时间。针对不同的应用目的可以使用不同的压缩方法。在数字图像处理领域中常用的编码有以下3种：

(1) 信息保持编码：该类编码技术主要应用于图像数字存储方面。该类编码技术要求能够最大限度的压缩图像大小，而且解码后能够无失真的恢复图像信息，通常也称该类编码为无误差编码。

(2) 保真度编码：该类编码主要应用于数字电视技术和静止图像通信方面。这些图像受传输信道容量的限制，接受图像信息的信宿又往往是人眼，过高的空间分辨率和过多的灰度层次不仅仅增加了数据量，且人眼无法接收。因此可以在编码过程中丢失一些对信宿无用或者用处不大的信息，也就是在允许失真的条件下和在一定的保真度准则下进行图像的压缩编码。

(3) 特征提取：在图像识别和分析、分类等技术中，往往并不需要全部的图像信息。例如，在文字识别过程中，不需要知道文字的具体灰度值，只要能够将文字和背景色区分开就可以了。因此，只要对需要的特征信息进行编码，就可以压缩图像数据量。显然，特征提取编码也是一种非信息保持编码。

压缩编码的具体方法很多，也有不同的分类方法，其中一种分类可以将图像编码分成以下四大类：

(1) 平均信息法

平均信息法编码是对每个像素单独处理，不考虑像素之间的相关性。在平均信息法编码

中常用的几种方法有：脉冲编码调制（Pulse Code Modulation，PCM）、熵编码（Entropy Coding）、行程编码（Run Length Coding）和位平面编码（Bit Plane Coding）。本章中将介绍熵编码中的哈夫曼（Huffman）编码以及行程编码（以读取.PCX 文件为例）。

(2) 预测编码法

预测编码是利用相邻像素之间的相关性，去掉图像中冗余的信息，只对有用的信息进行编码。举个简单的例子，由于像素的灰度是连续的，所以在一片区域中，相邻像素之间灰度值的差别可能很小。如果只记录第一个像素的灰度，其他像素的灰度都用它与前一个像素灰度之差来表示，就能起到压缩的目的。如表示灰度值为 250，253，251，252，252，250 的 6 个像素时，我们可以将它表示为：250，3，1，2，2，0。用原始的灰度值来保存需要 6×8＝48 个比特，而采用第二种表示方法，只需要 8＋2＋1＋2＋2＋1＝16 比特即可，这样就实现了压缩。常用的预测编码法有增量调制 ΔM（Delta Modulation，简称 DM）和微分预测编码（Differential Pulse Code Modulation，DPCM）。

(3) 变换编码法

是指将给定的图像变换到另一个数据域（如频域）上，使得大量的信息能用较少的数据来表示，从而达到压缩的目的的方法。常用的正交变换编码有前面介绍的离散傅立叶变换（Discrete Fourier Transform，DFT）、离散余弦变换（Discrete Cosine Transform，DCT）和离散沃尔什一哈达玛变换（Discrete Walsh-Hadamard Transform，DWHT)等。

(4) 其他编码法。

其他的编码方法也有很多，如内插法中的低取样和亚取样法，方块编码、混合编码（Hybird Coding）、矢量量化（Vector Quantize，VQ）和 LZW 算法等。在这里，我们只介绍 LZW 算法。

值得注意的是，近些年来出现了很多新的压缩编码方法，如使用人工神经元网络（Artificial Neural Network，ANN）的压缩编码算法、分形（Fractl）、小波（Wavelet）、基于对象（Object - Based）的压缩编码算法和基于模型（Model - Based）的压缩编码算法(应用在 MPEG4 及未来的视频压缩编码标准中)等等。

下面将介绍一些常用的图像编码方法。

11.1 哈夫曼编码

11.1.1 理论基础

哈夫曼（Huffman）编码是一种常用的压缩编码方法，是哈夫曼于 1952 年为压缩文本文件建立的。它的基本原理是频繁使用的数据用较短的代码代替，较少使用的数据用较长的代码代替，每个数据的代码各不相同。这些代码都是二进制码，且码的长度是可变的（因此，哈夫曼编码是一种变长编码方法）。

举个例子：假设一个副图像中出现了 8 种灰度级别：S_0，S_1，S_2，S_3，S_4，S_5，S_6和S_7，那么如果用等长的编码表示，每种灰度级别至少需要 3 比特，假设编码成 000，001，010，011，100，101，110，111（称做码字）。如果在一个像素序列中，S_0出现了 4 次，S_1出现了

5 次，S_2 出现了 6 次，S_3 出现了 7 次，S_4 出现了 10 次，S_5 出现了 10 次，S_6 出现了 18 次，S_7 出现了 40 次，按照上述编码后，一共要用 300 比特。

其实仔细观察一下该像素序列，可以发现 S_6 和 S_7 这两个符号出现的频率比较大，其他符号出现的频率比较小。如果我们采用一种编码方案使得 S_6 和 S_7 的码字短，其他符号的码字长，这样就能够减少占用的比特数。

例如，采用这样的编码方案：S_0 到 S_7 的码字分别为 00011，00010，0101，0100，0000，011，001，1，那么对上述图像编码只要用 261 比特。尽管有些码字变长了（如，S_0 由 3 位变成 5 位），但由于频繁使用的码字 S_7 变短了（从 3 位变成了 1 位），所以总体上长度变短了，从而实现了压缩。

上述的编码不是随意乱写出来的，它必须保证不能出现一个码字和另一个的前几位相同的情况。比如说，如果 S_0 的码字为 01，S_2 的码字为 011，那么当序列中出现 011 时，就不知道是 S_0 的码字后面跟了个 1，还是完整的一个 S_2 的码字。因此，必须给出的一个能保证这一点的编码算法。

下面给出具体的 Huffman 编码算法。

(1) 首先统计出每个灰度出现的频率，上例 S_0 到 S_7 的出现频率分别为 0.04，0.05，0.06，0.07，0.10，0.10，0.18 和 0.40。

(2) 从左到右把上述频率按从小到大的顺序排列。

(3) 每一次选出频率最小的两个值，将它们相加，形成的新频率值和其他频率值形成一个新的频率集合。

(4) 重复步骤 3，直到最后得到频率和为 1。如果用二叉树表示，上述过程如图 11－1 所示。

(5) 分配码字。将形成的二叉树的左节点标 1，右节点标 0（也可以全部反过来）。把从最上面的根节点到最下面的叶子节点途中遇到的 0,1 序列串起来，这样就得到了各个符号的编码。

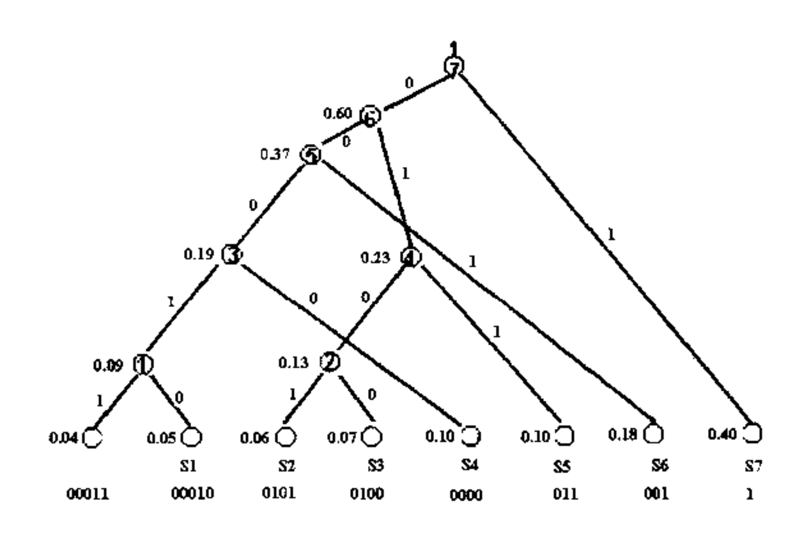


图 11－1 哈夫曼编码的示意图

在信息论中，可以定义图像的熵 H（Entropy）为：

$$H = -\sum_{i=0}^{N-1} P_i \log_2 P_i \tag{11-1}$$

其中 N 为图像的灰度级别，P_i 为第 i 级灰度出现的概率。

对于一种编码方法，设其第 i 级灰度码字的长度为 β_i 比特，那么可以定义平均码字长度 $\overline{N}$ 为：

$$\overline{N} = \sum_{i=0}^{N-1} \beta_i P_i \tag{11-2}$$

这样，就可以定义编码效率为：

$$\eta = \frac{H}{\overline{N}} \times 100\% \tag{11-3}$$

按照信息论中信源编码理论，编码效率 η 越大，表示编码方法越好。下面我们计算一下上述例子的编码效率：

$$\begin{aligned} H &= -\sum_{i=0}^{N-1} P_i \log_2 P_i \\ &= -0.04\log_2 0.04 - 0.05\log_2 0.05 - 0.06\log_2 0.06 - 0.07\log_2 0.07 \\ &\quad - 0.10\log_2 0.10 - 0.10\log_2 0.10 - 0.18\log_2 0.18 - 0.40\log_2 0.40 \\ &= 2.55 \end{aligned}$$

$$\begin{aligned} \overline{N} &= \sum_{i=0}^{N-1} \beta_i P_i \\ &= 0.04\times5 + 0.05\times5 + 0.06\times4 + 0.07\times4 + \\ &\quad 0.10\times4 + 0.10\times3 + 0.18\times3 + 0.40\times1 \\ &= 2.61 \end{aligned}$$

$$\eta = \frac{H}{\overline{N}} \times 100\% = \frac{2.55}{2.61} \times 100\% = 97.8\%$$

可见哈夫曼编码的编码效率是相当高的，其冗余度只有 2%。

在计算哈夫曼编码表时需要对原始图像数据扫描两遍：第一遍扫描要精确地统计出原始图像中每个灰度值出现的概率；第二遍是建立哈夫曼树并进行编码，由于需要建立二叉树并遍历二叉树生成编码，数据压缩和还原速度都较慢。但是该编码方法简单有效，而且编码效率相当高，因而应用非常广泛。

11.1.2 Visual C++实现哈夫曼编码

下面给出使用 Visual C++实现哈夫曼编码的程序。在这里，为了方便，只针对灰度进行

哈夫曼编码。对于其他的位图，也可以类似完成。

首先，给工程添加一个名为“图像编码”的菜单，并添加一个名为“哈夫曼编码”的菜单项。如图 11－2 所示。

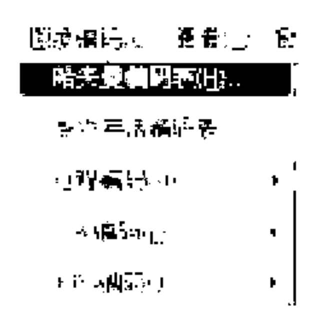


图 11－2 图像编码菜单

然后在该菜单事件中添加如下代码：

```
////////////////////////////////////////////////////////////////////////////////////////////
//  图像编码
//
void CCh1_1View::OnCodeHuffman()
{
    // 查看哈夫曼编码表

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向源图像像素的指针
    unsigned char * lpSrc;

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // DIB的高度
    LONG     lHeight;

    // DIB的宽度
    LONG     lWidth;

    // 图像每行的字节数
    LONG     lLineBytes;

    // 图像像素总数
    LONG     lCountSum;

    // 循环变量
    LONG     i;
```

```
LONG    j;

// 保存各个灰度值频率的数组指针
FLOAT * fFreq;

// 获取当前DIB颜色数目
int     iColorNum;

// 锁定DIB
lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

// 找到DIB图像像素起始位置
lpDIBBits = ::FindDIBBits(lpDIB);

// 获取当前DIB颜色数目
iColorNum = ::DIBNumColors(lpDIB);

// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图，其他的可以类推）
if (iColorNum != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图哈夫曼编码！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 更改光标形状
BeginWaitCursor();

/******************************************************************************
// 开始计算各个灰度级出现的频率
//
// 如果需要对其他序列进行哈夫曼编码，只需改动此处即可，例如，直接赋值：
   iColorNum = 8;
   fFreq = new FLOAT[iColorNum];
   fFreq[0] = 0.04;
   fFreq[1] = 0.05;
   fFreq[2] = 0.06;
   fFreq[3] = 0.07;
   fFreq[4] = 0.10;
   fFreq[5] = 0.10;
   fFreq[6] = 0.18;
   fFreq[7] = 0.40;
// 这样就可以对指定的序列进行哈夫曼编码
******************************************************************************/
```

```
// 分配内存
fFreq = new FLOAT[iColorNum];

// 计算DIB宽度
lWidth = ::DIBWidth(lpDIB);

// 计算DIB高度
lHeight = ::DIBHeight(lpDIB);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 重置计数为0
for (i = 0; i < iColorNum; i ++)
{
    // 清零
    fFreq[i] = 0.0;
}

// 计算各个灰度值的计数（对于非256色位图，此处给数组fFreq赋值方法将不同）
for (i = 0; i < lHeight; i ++)
{
    for (j = 0; j < lWidth; j ++)
    {
        // 指向图像指针
        lpSrc = (unsigned char *)lpDIBBits + lLineBytes * i + j;

        // 计数加1
        fFreq[*(lpSrc)] += 1;
    }
}

// 计算图像像素总数
lCountSum = lHeight * lWidth;

// 计算各个灰度值出现的概率
for (i = 0; i < iColorNum; i ++)
{
    // 计算概率
    fFreq[i] /= (FLOAT)lCountSum;
}

// 计算各个灰度级出现的频率结束
/*****************************************************************************/

// 创建对话框
CDlgHuffman dlgPara;

// 初始化变量值
dlgPara.m_fFreq = fFreq;
dlgPara.m_iColorNum = iColorNum;
```

```
    // 显示对话框
    dlgPara.DoModal();

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

}
```

其中 CdlgHuffman 是一个新创建的对话框类，该对话框主要功能是计算指定序列的哈夫曼编码表，同时计算图像熵、平均码字长度和编码效率。该对话框的完整代码如下。

1. 对话框头文件 DlgHuffman.h

```
#if !defined(AFX_DLGHUFFMAN_H__9BC92A31_5B8E_4A6D_B315_EE5ED22A2147__INCLUDED_)
#define AFX_DLGHUFFMAN_H__9BC92A31_5B8E_4A6D_B315_EE5ED22A2147__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgHuffman.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgHuffman dialog

class CDlgHuffman : public CDialog
{
// Construction
public:

    // 灰度级别数目
    int m_iColorNum;

    // 各个灰度值出现频率
    FLOAT * m_fFreq;

    // 哈夫曼编码表
    CString * m_strCode;

    CDlgHuffman(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgHuffman)
    enum { IDD = IDD_DLG_Huffman };
    CListCtrl   m_lstTable;
    double  m_dEntropy;
    double  m_dAvgCodeLen;
    double  m_dEfficiency;
```

```
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgHuffman)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgHuffman)
    virtual BOOL OnInitDialog();
    //}}AFX_MSG
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous
line.

#endif // !defined(AFX_DLGHUFFMAN_H__9BC92A31_5B8E_4A6D_B315_EE5ED22A2147__INCLUDED_)
```

2. 对话框头代码 DlgHuffman.cpp

```
// DlgHuffman.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgHuffman.h"
#include "DIBAPI.h"
#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgHuffman dialog


CDlgHuffman::CDlgHuffman(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgHuffman::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgHuffman)
    m_dEntropy = 0.0;
```

```
    m_dAvgCodeLen = 0.0;
    m_dEfficiency = 0.0;
    //}}AFX_DATA_INIT
}


void CDlgHuffman::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgHuffman)
    DDX_Control(pDX, IDC_LST_Table, m_lstTable);
    DDX_Text(pDX, IDC_EDIT1, m_dEntropy);
    DDX_Text(pDX, IDC_EDIT2, m_dAvgCodeLen);
    DDX_Text(pDX, IDC_EDIT3, m_dEfficiency);
    //}}AFX_DATA_MAP
}


BEGIN_MESSAGE_MAP(CDlgHuffman, CDialog)
    //{{AFX_MSG_MAP(CDlgHuffman)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgHuffman message handlers

BOOL CDlgHuffman::OnInitDialog()
{

    // 字符串变量
    CString str;

    // 循环变量
    LONG    i;
    LONG    j;
    LONG    k;

    // 中间变量
    FLOAT   fT;

    // ListCtrl的ITEM
    LV_ITEM lvitem;

    // 中间变量，保存ListCtrl中添加的ITEM编号
    int     iActualItem;

    // 调用默认得OnInitDialog()函数
    CDialog::OnInitDialog();

    // 初始化变量
    m_dEntropy = 0.0;
```

```
m_dAvgCodeLen = 0.0;

// 计算图像熵
for (i = 0; i < m_iColorNum; i ++)
{
    // 判断概率是否大于0
    if (m_fFreq[i] > 0)
    {
        // 计算图像熵
        m_dEntropy -= m_fFreq[i] * log(m_fFreq[i]) / log(2.0);
    }
}

// 保存计算中间结果的数组
FLOAT * fTemp;

// 保存映射关系的数组
int *   iMap;

// 分配内存
fTemp = new FLOAT[m_iColorNum];
iMap  = new int[m_iColorNum];
m_strCode = new CString[m_iColorNum];

// 初始化fTemp为m_fFreq
for (i = 0; i < m_iColorNum; i ++)
{
    // 赋值
    fTemp[i] = m_fFreq[i];
    iMap[i] = i;
}

// 用冒泡法对进行灰度值出现的概率排序，结果保存在数组fTemp中
for (j = 0; j < m_iColorNum - 1; j ++)
{
    for (i = 0; i < m_iColorNum - j - 1; i ++)
    {
        if (fTemp[i] > fTemp[i + 1])
        {
            // 互换
            fT = fTemp[i];
            fTemp[i] = fTemp[i + 1];
            fTemp[i + 1] = fT;

            // 更新映射关系
            for (k = 0; k < m_iColorNum; k ++)
            {
                // 判断是否是fTemp[i]的子节点
                if (iMap[k] == i)
                {
                    // 改变映射到节点i+1
```

```
                    iMap[k] = i + 1;
                }
                else if (iMap[k] == i + 1)
                {
                    // 改变映射到节点i
                    iMap[k] = i;
                }
            }
        }
    }
}

//////////////////////////////////////////////////////////////
// 计算哈夫曼编码表

// 找到概率大于0处才开始编码
for (i = 0; i < m_iColorNum - 1; i ++)
{
    // 判断概率是否大于0
    if (fTemp[i] > 0)
    {
        break;
    }
}

// 开始编码
for (; i < m_iColorNum - 1; i ++)
{
    // 更新m_strCode
    for (k = 0; k < m_iColorNum; k ++)
    {
        // 判断是否是fTemp[i]的子节点
        if (iMap[k] == i)
        {
            // 改变编码字符串
            m_strCode[k] = "1" + m_strCode[k];
        }
        else if (iMap[k] == i + 1)
        {
            // 改变编码字符串
            m_strCode[k] = "0" + m_strCode[k];
        }
    }

    // 概率最小的两个概率相加，保存在fTemp[i + 1]中
    fTemp[i + 1] += fTemp[i];

    // 改变映射关系
    for (k = 0; k < m_iColorNum; k ++)
    {
        // 判断是否是fTemp[i]的子节点
```

```
                if (iMap[k] == i)
                {
                    // 改变映射到节点i+1
                    iMap[k] = i + 1;
                }
            }

            // 重新排序
            for (j = i + 1; j < m_iColorNum - 1; j ++)
            {
                if (fTemp[j] > fTemp[j + 1])
                {
                    // 互换
                    fT = fTemp[j];
                    fTemp[j] = fTemp[j + 1];
                    fTemp[j + 1] = fT;

                    // 更新映射关系
                    for (k = 0; k < m_iColorNum; k ++)
                    {
                        // 判断是否是fTemp[i]的子节点
                        if (iMap[k] == j)
                        {
                            // 改变映射到节点j+1
                            iMap[k] = j + 1;
                        }
                        else if (iMap[k] == j + 1)
                        {
                            // 改变映射到节点j
                            iMap[k] = j;
                        }
                    }
                }
                else
                {
                    // 退出循环
                    break;
                }
            }
        }

        // 计算平均码字长度
        for (i = 0; i < m_iColorNum; i ++)
        {
            // 累加
            m_dAvgCodeLen += m_fFreq[i] * m_strCode[i].GetLength();
        }

        // 计算编码效率
        m_dEfficiency = m_dEntropy / m_dAvgCodeLen;
```

```
    // 保存变动
    UpdateData(FALSE);

    ///////////////////////////////////////////////////////////
    // 输出计算结果

    // 设置List控件样式
    m_lstTable.ModifyStyle(LVS_TYPEMASK, LVS_REPORT);

    // 给List控件添加Header
    m_lstTable.InsertColumn(0, "灰度值", LVCFMT_LEFT, 60, 0);
    m_lstTable.InsertColumn(1, "出现频率", LVCFMT_LEFT, 78, 0);
    m_lstTable.InsertColumn(2, "哈夫曼编码", LVCFMT_LEFT, 110, 1);
    m_lstTable.InsertColumn(3, "码字长度", LVCFMT_LEFT, 78, 2);

    // 设置样式为文本
    lvitem.mask = LVIF_TEXT;

    // 计算平均码字长度
    for (i = 0; i < m_iColorNum; i ++)
    {
        // 添加一项
        lvitem.iItem = m_lstTable.GetItemCount();
        str.Format("%u", i);
        lvitem.iSubItem = 0;
        lvitem.pszText= (LPTSTR)(LPCTSTR)str;
        iActualItem = m_lstTable.InsertItem(&lvitem);

        // 添加其他列
        lvitem.iItem = iActualItem;

        // 添加灰度值出现的频率
        lvitem.iSubItem = 1;
        str.Format("%f",m_fFreq[i]);
        lvitem.pszText = (LPTSTR)(LPCTSTR)str;
        m_lstTable.SetItem(&lvitem);

        // 添加哈夫曼编码
        lvitem.iSubItem = 2;
        lvitem.pszText = (LPTSTR)(LPCTSTR)m_strCode[i];
        m_lstTable.SetItem(&lvitem);

        // 添加码字长度
        lvitem.iSubItem = 3;
        str.Format("%u",m_strCode[i].GetLength());
        lvitem.pszText = (LPTSTR)(LPCTSTR)str;
        m_lstTable.SetItem(&lvitem);
    }
    // 返回TRUE
    return TRUE;
}
```

运行该代码，结果如图 11－3 所示：



图 11－3 图像哈夫曼编码结果

11.2 香农－弗诺编码

11.2.1 理论基础

香农－弗诺（Shannon－Fannon）编码也是一种常见的变长编码，利用该编码有时效率可以高达 100%。一般进行香农－费诺编码的步骤如下：

(1) 首先统计出每个灰度出现的频率。

(2) 从左到右把上述频率按从小到大的顺序排列。

(3) 从序列中某个位置将序列分成两个子序列，并尽量使两个序列频率和近似相等。给前面一个子序列赋值为 1，后面一个子序列赋值为 0。

(4) 重复步骤 3，直到各个子序列不能再分。

(5) 分配码字。将每个元素所属子序列的值串起来，这样就得到了各个元素的香农－弗诺编码。

下面举例来说明香农－弗诺编码具体过程。

例如，一幅图像有 8 种灰度级别：S_0，S_1，S_2，S_3，S_4，S_5，S_6和S_7，每种灰度出现的概率为 1/16，1/16，1/16，1/16，1/8，1/8，1/4，1/4。那么它的香农－弗诺编码过程如图 11－4

所示。

灰度值	概率					码字
S_7	$\frac{1}{4}$	0	0			00
S_6	$\frac{1}{4}$		1			01
S_5	$\frac{1}{8}$	1	0	0		100
S_4	$\frac{1}{8}$			1		101
S_3	$\frac{1}{16}$		1	0		1100
S_2	$\frac{1}{16}$					1101
S_1	$\frac{1}{16}$			1	0	1110
S_0	$\frac{1}{16}$				1	1111

图 11—4　香农—弗诺编码

下面我们来计算该中编码的编码效率：

$$
\begin{aligned}
H &= -\sum_{i=0}^{N-1} P_i \log_2 P_i \\
&= -\frac{1}{16}\times\log_2\frac{1}{16}\times4-\frac{1}{8}\times\log_2\frac{1}{8}\times2-\frac{1}{4}\times\log_2\frac{1}{4}\times2 \\
&= 2.75
\end{aligned}
$$

$$
\begin{aligned}
\overline{N} &= \sum_{i=0}^{N-1} \beta_i P_i \\
&= \frac{1}{16}\times4\times4+\frac{1}{8}\times3\times2+\frac{1}{4}\times2\times2 \\
&= 2.75
\end{aligned}
$$

$$
\eta=\frac{H}{\overline{N}}\times100\%=\frac{2.75}{2.75}\times100\%=100\%
$$

上述编码的效率高达 100%！其实如果各级灰度出现的概率正好为 2^{-N_i} 时，采用香农—弗诺编码进行编码，效率可以达到 100%；如果不满足该条件，编码效率没有那么高，但是结果还是令人满意的。

11.2.2 Visual C++编程实现

编程实现香农－弗诺编码比较简单，这里给出一个实现的源程序。首先在图像编码菜单中的香农－弗诺编码菜单项单击事件中添加如下代码：

```
void CCh1_1View::OnCodeShannon()
{
    // 查看香农－弗诺编码表

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向源图像像素的指针
    unsigned char * lpSrc;

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // DIB的高度
    LONG     lHeight;

    // DIB的宽度
    LONG     lWidth;

    // 图像每行的字节数
    LONG     lLineBytes;

    // 图像像素总数
    LONG     lCountSum;

    // 循环变量
    LONG     i;
    LONG     j;

    // 保存各个灰度值频率的数组指针
    FLOAT * fFreq;

    // 获取当前DIB颜色数目
    int      iColorNum;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 获取当前DIB颜色数目
    iColorNum = ::DIBNumColors(lpDIB);
```

```
// 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图，其他的可以类推）
if (iColorNum != 256)
{
    // 提示用户
    MessageBox("目前只支持256色位图香农－弗诺编码！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 返回
    return;
}

// 更改光标形状
BeginWaitCursor();

/*********************************************************************************
// 开始计算各个灰度级出现的频率
//
// 如果需要对其他序列进行香农－弗诺编码，只需改动此处即可，例如，直接赋值：
   iColorNum = 8;
   fFreq = new FLOAT[iColorNum];
   fFreq[0] = 0.0625;
   fFreq[1] = 0.0625;
   fFreq[2] = 0.0625;
   fFreq[3] = 0.0625;
   fFreq[4] = 0.125;
   fFreq[5] = 0.125;
   fFreq[6] = 0.25;
   fFreq[7] = 0.25;
// 这样就可以对指定的序列进行香农－弗诺编码
*********************************************************************************/

// 分配内存
fFreq = new FLOAT[iColorNum];

// 计算DIB宽度
lWidth = ::DIBWidth(lpDIB);

// 计算DIB高度
lHeight = ::DIBHeight(lpDIB);

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(lWidth * 8);

// 重置计数为0
for (i = 0; i < iColorNum; i ++)
{
    // 清零
    fFreq[i] = 0.0;
```

```
	}

	// 计算各个灰度值的计数（对于非256色位图，此处给数组fFreq赋值方法将不同）
	for (i = 0; i < lHeight; i ++)
	{
		for (j = 0; j < lWidth; j ++)
		{
			// 指向图像指针
			lpSrc = (unsigned char *)lpDIBBits + lLineBytes * i + j;

			// 计数加1
			fFreq[*(lpSrc)] += 1;
		}
	}

	// 计算图像像素总数
	lCountSum = lHeight * lWidth;

	// 计算各个灰度值出现的概率
	for (i = 0; i < iColorNum; i ++)
	{
		// 计算概率
		fFreq[i] /= (FLOAT)lCountSum;
	}

	// 计算各个灰度级出现的频率结束
	/*************************************************************************/

	// 创建对话框
	CDlgShannon dlgPara;

	// 初始化变量值
	dlgPara.m_fFreq = fFreq;
	dlgPara.m_iColorNum = iColorNum;

	// 显示对话框
	dlgPara.DoModal();

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();

}
```

其中 CdlgShannon 是一个新创建的对话框类，该对话框主要功能是计算指定序列的香农一费诺编码表，同时计算图像熵、平均码字长度和编码效率。该对话框的完整代码如下。

1. 对话框头文件 DlgShannon.h

```
#if !defined(AFX_DLGSHANNON_H__456A32D8_D7EE_4F4B_945B_672AE8258607__INCLUDED_)
```

```
#define AFX_DLGSHANNON_H__456A32D8_D7EE_4F4B_945B_672AE8258607__INCLUDED_

#if _MSC_VER > 1000
#pragma once
#endif // _MSC_VER > 1000
// DlgShannon.h : header file
//

/////////////////////////////////////////////////////////////////////////////
// CDlgShannon dialog

class CDlgShannon : public CDialog
{
// Construction
public:

    // 灰度级别数目
    int m_iColorNum;

    // 各个灰度值出现频率
    FLOAT * m_fFreq;

    // 香农－弗诺编码表
    CString * m_strCode;

    CDlgShannon(CWnd* pParent = NULL);   // standard constructor

// Dialog Data
    //{{AFX_DATA(CDlgShannon)
    enum { IDD = IDD_DLG_Shannon };
    CListCtrl   m_lstTable;
    double  m_dEntropy;
    double  m_dAvgCodeLen;
    double  m_dEfficiency;
    //}}AFX_DATA


// Overrides
    // ClassWizard generated virtual function overrides
    //{{AFX_VIRTUAL(CDlgShannon)
    protected:
    virtual void DoDataExchange(CDataExchange* pDX);    // DDX/DDV support
    //}}AFX_VIRTUAL

// Implementation
protected:

    // Generated message map functions
    //{{AFX_MSG(CDlgShannon)
    virtual BOOL OnInitDialog();
    //}}AFX_MSG
```

```
    DECLARE_MESSAGE_MAP()
};

//{{AFX_INSERT_LOCATION}}
// Microsoft Visual C++ will insert additional declarations immediately before the previous line.

#endif // !defined(AFX_DLGSHANNON_H__456A32D8_D7EE_4F4B_945B_672AE8258607__INCLUDED_)
```

2. 对话框头代码 DlgShannon.cpp

```
// DlgShannon.cpp : implementation file
//

#include "stdafx.h"
#include "ch1_1.h"
#include "DlgShannon.h"
#include <math.h>

#ifdef _DEBUG
#define new DEBUG_NEW
#undef THIS_FILE
static char THIS_FILE[] = __FILE__;
#endif

/////////////////////////////////////////////////////////////////////////////
// CDlgShannon dialog


CDlgShannon::CDlgShannon(CWnd* pParent /*=NULL*/)
    : CDialog(CDlgShannon::IDD, pParent)
{
    //{{AFX_DATA_INIT(CDlgShannon)
    m_dEntropy = 0.0;
    m_dAvgCodeLen = 0.0;
    m_dEfficiency = 0.0;
    //}}AFX_DATA_INIT
}


void CDlgShannon::DoDataExchange(CDataExchange* pDX)
{
    CDialog::DoDataExchange(pDX);
    //{{AFX_DATA_MAP(CDlgShannon)
    DDX_Control(pDX, IDC_LST_Table, m_lstTable);
    DDX_Text(pDX, IDC_EDIT1, m_dEntropy);
    DDX_Text(pDX, IDC_EDIT2, m_dAvgCodeLen);
    DDX_Text(pDX, IDC_EDIT3, m_dEfficiency);
    //}}AFX_DATA_MAP
}
```

```
BEGIN_MESSAGE_MAP(CDlgShannon, CDialog)
    //{{AFX_MSG_MAP(CDlgShannon)
    //}}AFX_MSG_MAP
END_MESSAGE_MAP()

/////////////////////////////////////////////////////////////////////////////
// CDlgShannon message handlers

BOOL CDlgShannon::OnInitDialog()
{

    // 字符串变量
    CString str;

    // 循环变量
    LONG    i;
    LONG    j;

    // 中间变量
    FLOAT   fT;
    LONG    iTemp;

    // 保存计算中间结果的数组
    FLOAT * fTemp;

    // 保存映射关系的数组
    LONG *  iMap;

    // 当前编码区间的频率和
    FLOAT   fTotal;

    // 计数（编码完成的个数）
    LONG    iCount;

    // 频率和
    FLOAT   fSum;

    // 起始位置
    LONG    iStart;

    // 指向布尔型数组的指针
    BOOL    * bFinished;

    // 调用默认得OnInitDialog()函数
    CDialog::OnInitDialog();

    // 初始化变量
    m_dEntropy = 0.0;
    m_dAvgCodeLen = 0.0;

    // 计算图像熵
```

```
for (i = 0; i < m_iColorNum; i ++)
{
    // 判断概率是否大于0
    if (m_fFreq[i] > 0)
    {
        // 计算图像熵
        m_dEntropy -= m_fFreq[i] * log(m_fFreq[i]) / log(2.0);
    }
}

// 分配内存
fTemp = new FLOAT[m_iColorNum];
m_strCode = new CString[m_iColorNum];
bFinished = new BOOL[m_iColorNum];
iMap  = new LONG[m_iColorNum];

fTotal = 0;

// 初始化fTemp为m_fFreq, bFinished为FALSE
for (i = 0; i < m_iColorNum; i ++)
{
    // 赋值
    fTemp[i] = m_fFreq[i];

    // 初始化映射关系
    iMap[i] = i;

    // 初始化为FALSE
    bFinished[i] = FALSE;

    // 计算fTotal
    fTotal += m_fFreq[i];
}

// 用冒泡法对进行灰度值出现的概率排序，结果保存在数组fTemp中
for (j = 0; j < m_iColorNum - 1; j ++)
{
    for (i = 0; i < m_iColorNum - j - 1; i ++)
    {
        if (fTemp[i] > fTemp[i + 1])
        {
            // 互换
            fT = fTemp[i];
            fTemp[i] = fTemp[i + 1];
            fTemp[i + 1] = fT;

            // 更新映射关系
            iTemp = iMap[i];
            iMap[i] = iMap[i+1];
            iMap[i+1] = iTemp;
```

```
        }
    }
}

////////////////////////////////////////////////////////////
// 计算香农－弗诺编码表

// 找到概率大于0处才开始编码
for (iStart = 0; iStart < m_iColorNum - 1; iStart ++)
{
    // 判断概率是否大于0
    if (fTemp[iStart] > 0)
    {
        // 跳出
        break;
    }
}

// 初始化变量
fSum = 0;
str = "1";

// 开始编码
while(iCount < m_iColorNum)
{
    // 初始化iCount为iStart
    iCount = iStart;

    // 循环编码
    for (i = iStart; i < m_iColorNum; i ++)
    {
        // 判断是否编码完成
        if (bFinished[i] == FALSE)
        {
            // 编码没有完成，继续编码

            // fSum加当前出现的频率
            fSum += fTemp[i];

            // 判断是否超出总和的一半
            if (fSum > fTotal/2.0)
            {
                // 超出，追加的字符改为0
                str = "0";
            }

            // 编码追加字符1或0
            m_strCode[iMap[i]] += str;

            // 判断是否编码完一段
            if (fSum == fTotal)
```

```
    {
        // 完成一部分编码，重新计算fTotal

        // 初始化fSum为0
        fSum = 0;

        // 判断是否是最后一个元素
        if (i == m_iColorNum - 1)
        {
            // 是最后，设置从起始点开始
            j = iStart;
        }
        else
        {
            // 不是最后，设置从下一个点开始
            j = i + 1;
        }

        // 保存j值
        iTemp = j;
        str = m_strCode[iMap[j]];

        // 计算下一段的fTotal
        fTotal = 0;
        for (; j < m_iColorNum; j++)
        {
            // 判断是否是同一段编码
            if ((m_strCode[iMap[j]].Right(1) != str.Right(1))
                || (m_strCode[iMap[j]].GetLength() != str.GetLength()))
            {
                // 退出循环
                break;
            }

            // 累加
            fTotal += fTemp[j];
        }

        // 初始化str为1
        str = "1";

        // 判断是否该段长度为1
        if (iTemp + 1 == j)
        {
            // 是，表示该段编码已经完成
            bFinished[iTemp] = TRUE;
        }
    }
}
else
{
```

```
            // iCount加1
            iCount ++;

            // 计算下一次循环的fTotal

            // 初始化fSum为0
            fSum = 0;

            // 判断是否是最后一个元素
            if (i == m_iColorNum - 1)
            {
                // 是最后，设置从起始点开始
                j = iStart;
            }
            else
            {
                // 不是最后，设置从下一个点开始
                j = i + 1;
            }

            // 保存j值
            iTemp = j;
            str = m_strCode[iMap[j]];

            // 计算下一段的fTotal
            fTotal = 0;
            for (; j < m_iColorNum; j++)
            {
                // 判断是否是同一段编码
                if ((m_strCode[iMap[j]].Right(1) != str.Right(1))
                    || (m_strCode[iMap[j]].GetLength() != str.GetLength()))
                {
                    // 退出循环
                    break;
                }

                // 累加
                fTotal += fTemp[j];
            }

            // 初始化str为1
            str = "1";

            // 判断是否该段长度为1
            if (iTemp + 1 == j)
            {
                // 是，表示该段编码已经完成
                bFinished[iTemp] = TRUE;
            }
        }
    }
```

```
}

// 计算平均码字长度
for (i = 0; i < m_iColorNum; i ++)
{
    // 累加
    m_dAvgCodeLen += m_fFreq[i] * m_strCode[i].GetLength();
}

// 计算编码效率
m_dEfficiency = m_dEntropy / m_dAvgCodeLen;

// 保存变动
UpdateData(FALSE);

//////////////////////////////////////////////////////////
// 输出计算结果

// ListCtrl的ITEM
LV_ITEM lvitem;

// 中间变量，保存ListCtrl中添加的ITEM编号
int     iActualItem;

// 设置List控件样式
m_lstTable.ModifyStyle(LVS_TYPEMASK, LVS_REPORT);

// 给List控件添加Header
m_lstTable.InsertColumn(0, "灰度值", LVCFMT_LEFT, 60, 0);
m_lstTable.InsertColumn(1, "出现频率", LVCFMT_LEFT, 78, 0);
m_lstTable.InsertColumn(2, "香农弗诺编码", LVCFMT_LEFT, 110, 1);
m_lstTable.InsertColumn(3, "码字长度", LVCFMT_LEFT, 78, 2);

// 设置样式为文本
lvitem.mask = LVIF_TEXT;

// 计算平均码字长度
for (i = 0; i < m_iColorNum; i ++)
{
    // 添加一项
    lvitem.iItem = m_lstTable.GetItemCount();
    str.Format("%u", i);
    lvitem.iSubItem = 0;
    lvitem.pszText= (LPTSTR)(LPCTSTR)str;
    iActualItem = m_lstTable.InsertItem(&lvitem);

    // 添加其他列
    lvitem.iItem = iActualItem;

    // 添加灰度值出现的频率
    lvitem.iSubItem = 1;
```

```
        str.Format("%f",m_fFreq[i]);
        lvitem.pszText = (LPTSTR)(LPCTSTR)str;
        m_lstTable.SetItem(&lvitem);

        // 添加香农弗诺编码
        lvitem.iSubItem = 2;
        lvitem.pszText = (LPTSTR)(LPCTSTR)m_strCode[i];
        m_lstTable.SetItem(&lvitem);

        // 添加码字长度
        lvitem.iSubItem = 3;
        str.Format("%u",m_strCode[i].GetLength());
        lvitem.pszText = (LPTSTR)(LPCTSTR)str;
        m_lstTable.SetItem(&lvitem);
    }

    // 返回TRUE
    return TRUE;
}
```

运行该代码，结果如图 11－5 所示。

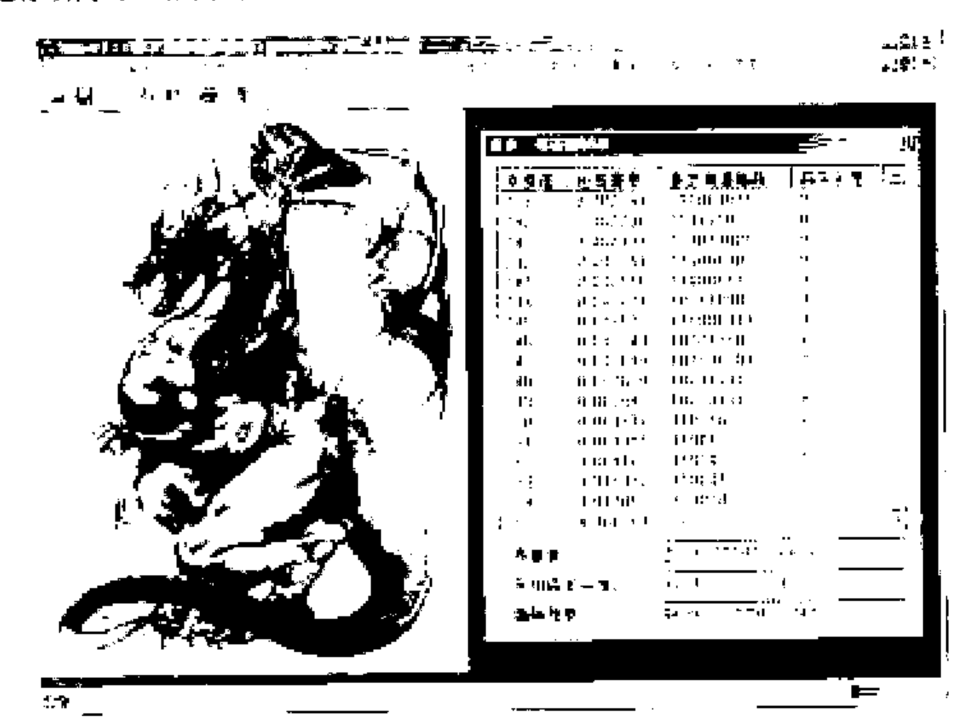


图 11－5　图像香农－费诺编码结果

11.3 行程编码

11.3.1 理论基础

行程编码（Run Length Coding）的原理十分简单：将一行中颜色值相同的相邻像素用一个计数值和该颜色值来代替。例如：aaabccccccddeee 可以表示为 3a1b6c2d3e。如果一幅图像是由很多块颜色相同的大面积区域组成的，那么采用行程编码的压缩效率是惊人的。

然而，该算法也存在一个致命弱点：如果图像中每两个相邻点的颜色都不同，用这种算法不但不能压缩，反而数据量增加一倍。所以现在单纯采用行程编码的压缩算法用得并不多，PCX 文件是其中的一种。下面我们就以 PCX 文件为例来介绍行程编码。

11.3.2 PCX 文件格式及其编码方法

PCX（也称 PCC）文件格式最早是由 zSoft 公司开发的 PC Paintbrush 软件所采用的，由于 Paintbrush 曾经是 Microsoft Mouse 的附属品，而且 PCX 文件格式简单易懂，编程非常容易实现，几乎所有的图像应用软件都支持该文件格式。因此 PCX 早就成为众所皆知的图像文件格式。但是由于 PCX 格式压缩比不高，其文件格式也不太合理，现在用得并不是很多了。

PCX 文件结构非常简单，仅分为文件头和图像压缩数据两个部分（如果是 256 色 PCX 图像文件，则还有一个 256 色调色板存于文件尾部）。下面将分别进行说明。

(1) PCX 文件头全长 128 字节，其数据结构如下：

```
typedef struct{
        BYTE bManufacturer;
        BYTE bVersion;
        BYTE bEncoding;
        BYTE bBpp;
        WORD wLeft;
        WORD wTop;
        WORD wRight;
        WORD wBottom;
        WORD wXResolution;
        WORD wYResolution;
        BYTE bPalette[48];
        BYTE bReserved;
        BYTE bPlanes;
        WORD wLineBytes;
        WORD wPaletteType;
        WORD wSrcWidth;
        WORD wSrcDepth;
        BYTE bFiller[54];
} PCXHEADER;
```

其中：

➢ bManufacturer 为 PCX 文件的标识，必须为 0x0A，可以通过该域来判断一幅图像是否是 PCX 图像文件；

➢ bVersion 域指明当前 PCX 文件的版本号，其值含义如表 11－1 所示。

表 11－1　　bVersion 取值表

Version 取值	对应的 Painbrush 版本	支持的色彩
0	PC Painbrush 2.5	单色和 4 色
2	PC Painbrush 2.8（包含调色板数据）	单色、4 色、8 色和 16 色
3	PC Painbrush 2.8（不包含调色板数据，采用系统默认调色板）	单色、4 色、8 色和 16 色
4	PC Painbrush for Window	单色、4 色、8 色和 16 色
5	PC Painbrush 3.0、PC Painbrush 4.0（Plus）、PC Painbrush Plus for Window	单色、4 色、8 色、16 色、256 色和真彩色

➢ bEncoding 域目前取值固定为 1，表示采用行程编码。如果未来的 PCX 加入新的编码方式，则应该通过该域来判断数据的编码方式。

➢ bBpp 指明每个像素所需要的位数。

➢ wLeft 指明图像相对于屏幕的左上角 x 坐标（以像素为单位）。

➢ wTop 指明图像相对于屏幕的左上角 y 坐标（以像素为单位）。

➢ wRight 指明图像相对于屏幕的右下角 x 坐标（以像素为单位）。

➢ wBottom 指明图像相对于屏幕的右下角 y 坐标（以像素为单位）。

图像的宽度为 wRight−wLeft + 1，同样图像的高度为 wBottom−wTop + 1。

➢ wXResolution 域指明图像的水平分辨率（每英寸有多少个像素）。

➢ wYResolution 域指明图像的垂直分辨率（每英寸有多少个像素）。通过 wXResolution 和 wYResolution 可以使图像在输出设备上有最佳效果。

➢ bPalette 域指明调色板数据。由于该域长度为 48 字节，只能保存 16 种颜色（RGB 值，每种颜色需要 3 字节）。这样的调色板不可能保存 256 色（需要 256×3＝768 字节），因此，对应 256 色图像，它的调色板保存在图像的尾部，此时 bPalette 域是没有任何意义的（浪费了 48 字节，可见制定 PCX 文件结构时缺乏远见）。

➢ bReserved 为保留域，设定为 0。

➢ bPlanes 指明图像色彩平面数目。该域和 bBpp 域决定图像的颜色总数。对应的组合方式如表 11－2 所示。

表 11－2　　PCX 文件色彩数目

bBpp 取值	bPlanes 取值	色彩数目
1	1	单色
1	4	16 色
8	1	256 色
8	3	真彩色

➢ wLineBytes 域指定图像的宽度（字节为单位），它必须为偶数。

➢ wPalatteType 域指定图像调色板的类型，1 表示彩色或者单色图像，2 表示图像是灰度图。

➢ wSrcWidth 域指定制作该图像的屏幕宽度（像素为单位，0 为基准，即取值为屏幕

宽度减 1)。

➢ wSrcDepth 域指定制作该图像的屏幕高度(像素为单位,0 为基准,即取值为屏幕高度减 1)。

➢ bFiller 域也是保留域,取值设定为 0。

(2) 图像压缩数据

图像压缩数据紧跟在文件头后面。图像数据存储和图像颜色数目是紧密相关的。这里为了更加方便地介绍行程编码,只介绍 256 色 PCX 文件。对于其他色彩的 PCX 文件读者可以查阅相关的图像文件格式的书籍。

256 色 PCX 文件每个像素一个字节,编码时按照从左到右从上到下的顺序进行(如果图像的宽度为奇数,那么每行需要添加一个填充字节)。进行编码时,是以字节为单位,一行一行地进行。首先计算原始数据中各个数据出现的次数,然后用该数据重复次数加上数据本身来代替原始数据。编码原则如下:

➢ 图像数据以字节为单位进行编码的。

➢ 对于连续重复的像素值,统计其连续出现的次数 iCount(最大取值为 63),先存入长度信息(iCount | 0xC0),然后再存入像素值。如果连续次数超过 63 次,则必须分多次处理。例如,连续 132 个 0x98,编码时必须分三次处理,编码结果为:0xFF 0x98 0xFF 0x98 0xC6 0x98。

➢ 如果遇到不重复的像素值,如果该像素值小于等于 0xC0,则直接存入该像素值。否则首先存入一个 0xC1,然后再存入该像素值。这样做是为了避免该像素值被误认为是数据长度。

有了上述的编码原则,那么解码时也很容易。首先读取一个字节到 bChar 中,判断该字节是否大于 0xC0,如果是则表明是行程(Run Length)信息,即 bChar 的低 6 位表示后面连续的字节个数,保存在变量 iCount 中,读取下一个字节并重复 iCount 次存入图像像素缓冲区;否则直接将 bChar 存入图像像素缓冲区。

(3) 256 色调色板

对于 256 色 PCX 文件,在图像数据后还有一个长为 769 字节的 256 色调色板。其中它的第一个字节为调色板标志字节,取值恒定为 0x0C。接下来的 768(256×3)字节为调色板的内容,即 256 种颜色的 RGB 值。

11.3.3 编程实现 PCX 文件的读写

了解了 PCX 文件的格式和行程编码的原则,那么编程实现读写 PCX 功能应该比较容易了。首先必须编写一个行程编码程序,该程序可以用来将 DIB 文件转化为 PCX 文件。下面的函数 DIBToPCX256()就可以将指定的 DIB 文件保存为 PCX 格式文件。

```
/*************************************************************************
 *
 * 函数名称:
 *   DIBToPCX256()
 *
 * 参数:
 *   LPSTR lpDIB          - 指向DIB对象的指针
```

```
 *   CFile& file        - 要保存的文件
 *
 * 返回值:
 *   BOOL               - 成功返回True，否则返回False。
 *
 * 说明:
 *   该函数将指定的256色DIB对象保存为256色PCX文件。
 *
 ************************************************************************/
BOOL WINAPI DIBToPCX256(LPSTR lpDIB, CFile& file)
{
    // 循环变量
    LONG    i;
    LONG    j;

    // DIB高度
    WORD    wHeight;

    // DIB宽度
    WORD    wWidth;

    // 中间变量
    BYTE    bChar1;
    BYTE    bChar2;

    // 指向原图像像素的指针
    BYTE *  lpSrc;

    // 指向编码后图像数据的指针
    BYTE *  lpDst;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 重复像素计数
    int     iCount;

    // 缓冲区已使用的字节数
    DWORD   dwBuffUsed;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // 获取DIB高度
    wHeight = (WORD) DIBHeight(lpDIB);

    // 获取DIB宽度
    wWidth  = (WORD) DIBWidth(lpDIB);

    // 找到DIB图像像素起始位置
    lpDIBBits = FindDIBBits(lpDIB);
```

```
// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(wWidth * 8);


//************************************************************************
// PCX文件头
PCXHEADER pcxHdr;

// 给文件头赋值

// PCX标识码
pcxHdr.bManufacturer = 0x0A;

// PCX版本号
pcxHdr.bVersion = 5;

// PCX编码方式（1表示RLE编码）
pcxHdr.bEncoding = 1;

// 像素位数（256色为8位）
pcxHdr.bBpp = 8;

// 图像相对于屏幕的左上角X坐标（以像素为单位）
pcxHdr.wLeft = 0;

// 图像相对于屏幕的左上角Y坐标（以像素为单位）
pcxHdr.wTop = 0;

// 图像相对于屏幕的右下角X坐标（以像素为单位）
pcxHdr.wRight = wWidth - 1;

// 图像相对于屏幕的右下角Y坐标（以像素为单位）
pcxHdr.wBottom = wHeight - 1;

// 图像的水平分辨率
pcxHdr.wXResolution = wWidth;

// 图像的垂直分辨率
pcxHdr.wYResolution = wHeight;

// 调色板数据（对于256色PCX无意义，直接赋值为0）
for (i = 0; i < 48; i ++)
{
    pcxHdr.bPalette[i] = 0;
}

// 保留域，设定为0。
pcxHdr.bReserved = 0;

// 图像色彩平面数目（对于256色PCX设定为1）。
```

```
pcxHdr.bPlanes = 1;

// 图像的宽度（字节为单位），必须为偶数。
if ((wWidth & 1) == 0)
{
    pcxHdr.wLineBytes = wWidth;
}
else
{
    pcxHdr.wLineBytes = wWidth + 1;
}

// 图像调色板的类型，1表示彩色或者单色图像，2表示图像是灰度图。
pcxHdr.wPaletteType = 1;

// 制作该图像的屏幕宽度（像素为单位）
pcxHdr.wSrcWidth = 0;

// 制作该图像的屏幕高度（像素为单位）
pcxHdr.wSrcDepth = 0;

// 保留域，取值设定为0。
for (i = 0; i < 54; i ++)
{
    pcxHdr.bFiller[i] = 0;
}

// 写入文件头
file.Write((LPSTR)&pcxHdr, sizeof(PCXHEADER));

//**************************************************************************************
// 开始编码

// 开辟一片缓冲区(2被原始图像大小)以保存编码结果
lpDst = new BYTE[wHeight * wWidth * 2];

// 指明当前已经用了多少缓冲区（字节数）
dwBuffUsed = 0;

// 每行
for (i = 0; i < wHeight; i++)
{
    // 指向DIB第i行，第0个像素的指针
    lpSrc = (BYTE *)lpDIBBits + lLineBytes * (wHeight - 1 - i);

    // 给bChar1赋值
    bChar1 = *lpSrc;

    // 设置iCount为1
    iCount = 1;
```

```
// 剩余列
for (j = 1; j < wWidth; j ++)
{
    // 指向DIB第i行，第j个像素的指针
    lpSrc++;

    // 读取下一个像素
    bChar2 = *lpSrc;

    // 判断是否和bChar1相同并且iCount < 63
    if ((bChar1 == bChar2) && (iCount < 63))
    {
        // 相同，计数加1
        iCount ++;

        // 继续读下一个
    }
    else
    {
        // 不同，或者iCount = 63

        // 写入缓冲区
        if ((iCount > 1) || (bChar1 >= 0xC0))
        {
            // 保存码长信息
            lpDst[dwBuffUsed] = iCount | 0xC0;

            // 保存bChar1
            lpDst[dwBuffUsed + 1] = bChar1;

            // 更新dwBuffUsed
            dwBuffUsed += 2;
        }
        else
        {
            // 直接保存该值
            lpDst[dwBuffUsed] = bChar1;

            // 更新dwBuffUsed
            dwBuffUsed ++;
        }

        // 重新给bChar1赋值
        bChar1 = bChar2;

        // 设置iCount为1
        iCount = 1;
    }
}

// 保存每行最后一部分编码
```

```
        if ((iCount > 1) || (bChar1 >= 0xC0))
        {
            // 保存码长信息
            lpDst[dwBuffUsed] = iCount | 0xC0;

            // 保存bChar1
            lpDst[dwBuffUsed + 1] = bChar1;

            // 更新dwBuffUsed
            dwBuffUsed += 2;
        }
        else
        {
            // 直接保存该值
            lpDst[dwBuffUsed] = bChar1;

            // 更新dwBuffUsed
            dwBuffUsed ++;
        }
    }

    // 写入编码结果
    file.WriteHuge((LPSTR)lpDst, dwBuffUsed);

    // 释放内存
    delete lpDst;

    //**************************************************************************
    // 写入调色板信息

    // 指向BITMAPINFO结构的指针（Win3.0）
    LPBITMAPINFO lpbmi;

    // 指向BITMAPCOREINFO结构的指针
    LPBITMAPCOREINFO lpbmc;

    // 表明是否是Win3.0 DIB的标记
    BOOL bWinStyleDIB;

    // 开辟一片缓冲区以保存调色板
    lpDst = new BYTE[769];

    // 调色板起始字节
    * lpDst = 0x0C;

    // 获取指向BITMAPINFO结构的指针（Win3.0）
    lpbmi = (LPBITMAPINFO)lpDIB;

    // 获取指向BITMAPCOREINFO结构的指针
    lpbmc = (LPBITMAPCOREINFO)lpDIB;
```

```
    // 判断是否是WIN3.0的DIB
    bWinStyleDIB = IS_WIN30_DIB(lpDIB);

    // 读取当前DIB调色板
    for (i = 0; i < 256; i ++)
    {
        if (bWinStyleDIB)
        {
            // 读取DIB调色板红色分量
            lpDst[i * 3 + 1] = lpbmi->bmiColors[i].rgbRed;

            // 读取DIB调色板绿色分量
            lpDst[i * 3 + 2] = lpbmi->bmiColors[i].rgbGreen;

            // 读取DIB调色板蓝色分量
            lpDst[i * 3 + 3] = lpbmi->bmiColors[i].rgbBlue;
        }
        else
        {
            // 读取DIB调色板红色分量
            lpDst[i * 3 + 1] = lpbmc->bmciColors[i].rgbtRed;

            // 读取DIB调色板绿色分量
            lpDst[i * 3 + 2] = lpbmc->bmciColors[i].rgbtGreen;

            // 读取DIB调色板蓝色分量
            lpDst[i * 3 + 3] = lpbmc->bmciColors[i].rgbtBlue;
        }
    }

    // 写入调色板信息
    file.Write((LPSTR)lpDst, 769);

    // 返回
    return TRUE;
}
```

利用上面的函数可以将当前打开的图像存成 PCX 格式。下面我们来编写菜单“行程编码”子菜单中的“保存成 PCX 文件”菜单项（如图 11—6 所示）的单击事件。

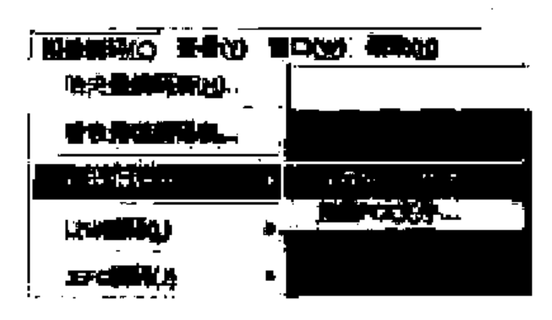


图 11—6 行程编码菜单

```
void CCh1_1View::OnCodeRLE()
{
    // 对当前图像进行行程编码（存为PCX格式文件）

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR   lpDIB;

    // 指向DIB像素指针
    LPSTR    lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否是8-bpp位图（这里为了方便，只处理8-bpp位图的行程编码）
    if (::DIBNumColors(lpDIB) != 256)
    {
        // 提示用户
        MessageBox("目前只支持256色位图的行程编码！", "系统提示" ,
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 文件保存路径
    CString strFilePath;

    // 初始化文件名为原始文件名
    strFilePath = pDoc->GetPathName();

    // 更改后缀为PCX
    if (strFilePath.Right(4).CompareNoCase(".BMP") == 0)
    {
        // 更改后缀为PCX
        strFilePath = strFilePath.Left(strFilePath.GetLength()-3) + "PCX";
    }
    else
    {
        // 直接添加后缀PCX
```

```
    strFilePath += ".PCX";
}

// 创建SaveAs对话框
CFileDialog dlg(FALSE, "PCX", strFilePath, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
    "PCX图像文件 (*.PCX) | *.PCX|所有文件 (*.*) | *.*||", NULL);

// 提示用户选择保存的路径
if (dlg.DoModal() != IDOK)
{
    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

    // 返回
    return;
}

// 获取用户指定的文件路径
strFilePath = dlg.GetPathName();

// CFile和CFileException对象
CFile file;
CFileException fe;

// 尝试创建指定的PCX文件
if (!file.Open(strFilePath, CFile::modeCreate |
  CFile::modeReadWrite | CFile::shareExclusive, &fe))
{
    // 提示用户
    MessageBox("打开指定PCX文件时失败！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 返回
    return;
}

// 调用DIBToPCX256()函数将当前的DIB保存为256色PCX文件
if (::DIBToPCX256(lpDIB, file))
{
    // 提示用户
    MessageBox("成功保存为PCX文件！", "系统提示",
        MB_ICONINFORMATION | MB_OK);
}
else
{
    // 提示用户
    MessageBox("保存为PCX文件失败！", "系统提示",
        MB_ICONINFORMATION | MB_OK);
```

```
	}

	// 解除锁定
	::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

	// 恢复光标
	EndWaitCursor();
}
```

运行上述代码，如果打开一幅 256 色的 BMP 文件，可以选择该菜单项将 BMP 文件保存为 PCX 文件。图 11－7 所示的 BMP 文件原始大小为 172,800 字节，如果用上面的代码将它保存为 PCX 格式，则文件大小为 140,610 字节。可见行程编码虽然能够进行图像压缩，但其编码效率不是很理想。



图 11－7　原始 BMP 图像

✍ **对于上面的灰度图像如果我们将它的调色板逆转，即调色板中第一个颜色为（255, 255, 255），第二种颜色为（254, 254, 254），…，第 256 种颜色为（0, 0, 0），同时更新图像像素在调色板中的颜色索引（直接拿 255 减去以前的索引值即可），图像外观没有发生任何变化。但是，在进行行程编码时，由于图像中大多数颜色灰度值较高，更改调色板后索引值变小（255 减去灰度值），小于 0xC0 的索引值会比以前多，那么变换后的图像进行 PCX 行程编码结果要好些。实际上也的确如此。变换后再编码的文件大小为 118,822 字节。编码效率又进一步提高。用 Photoshop 6.0 将 BMP 文件另存成 PCX 文件时就会采用上述的方法进行优化。**

在读取 PCX 文件时，首先要将编码后的结果解码。解码的思路正好和编码相反，首先读取一个字节到 bChar 中，判断该字节是否大于 0xC0，如果是则表明是行程（Run Length）信息，即 bChar 的低 6 位表示后面连续的字节个数，保存在变量 iCount 中，读取下一个字节并

重复 iCount 次存入图像像素缓冲区；否则直接将 bChar 存入图像像素缓冲区。下面的 ReadPCX256()函数就能够将一幅 256 色的 PCX 文件转化成 DIB（BMP）文件格式。

```
/*************************************************************************
 *
 * 函数名称：
 *   ReadPCX256()
 *
 * 参数：
 *   CFile& file        - 要读取的文件
 *
 * 返回值：
 *   HDIB               - 成功返回DIB的句柄，否则返回NULL。
 *
 * 说明：
 *   该函数将读取指定的256色PCX文件。将读取的结果保存在一个未压缩
 * 编码的DIB对象中。
 *
 *************************************************************************/
HDIB WINAPI ReadPCX256(CFile& file)
{
    // PCX文件头
    PCXHEADER pcxHdr;

    // DIB大小（字节数）
    DWORD   dwDIBSize;

    // DIB句柄
    HDIB    hDIB;

    // DIB指针
    LPSTR   pDIB;

    // 循环变量
    LONG    i;
    LONG    j;

    // 重复像素计数
    int     iCount;

    // DIB高度
    WORD    wHeight;

    // DIB宽度
    WORD    wWidth;

    // 图像每行的字节数
    LONG    lLineBytes;

    // 中间变量
    BYTE    bChar;
```

```
// 指向源图像像素的指针
BYTE *  lpSrc;

// 指向编码后图像数据的指针
BYTE *  lpDst;

// 临时指针
BYTE *  lpTemp;

// 尝试读取PCX文件头
if (file.Read((LPSTR)&pcxHdr, sizeof(PCXHEADER)) != sizeof(PCXHEADER))
{
    // 大小不对，返回NULL。
    return NULL;
}

// 判断是否是256色PCX文件，检查第一个字节是否是0x0A,
if ((pcxHdr.bManufacturer != 0x0A) || (pcxHdr.bBpp != 8) || (pcxHdr.bPlanes != 1))
{
    // 非256色PCX文件，返回NULL。
    return NULL;
}

// 获取图像高度
wHeight = pcxHdr.wBottom - pcxHdr.wTop + 1;

// 获取图像宽度
wWidth  = pcxHdr.wRight - pcxHdr.wLeft + 1;

// 计算图像每行的字节数
lLineBytes = WIDTHBYTES(wWidth * 8);

// 计算DIB长度（字节）
dwDIBSize = sizeof(BITMAPINFOHEADER) + 1024 + wHeight * lLineBytes;

// 为DIB分配内存
hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, dwDIBSize);
if (hDIB == 0)
{
    // 内存分配失败，返回NULL。
    return NULL;
}

// 锁定
pDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

// 指向BITMAPINFOHEADER的指针
LPBITMAPINFOHEADER lpBI;

// 赋值
```

```
lpBI = (LPBITMAPINFOHEADER) pDIB;

// 给lpBI成员赋值
lpBI->biSize = 40;
lpBI->biWidth = wWidth;
lpBI->biHeight = wHeight;
lpBI->biPlanes = 1;
lpBI->biBitCount = 8;
lpBI->biCompression = BI_RGB;
lpBI->biSizeImage = wHeight * lLineBytes;
lpBI->biXPelsPerMeter = pcxHdr.wXResolution;
lpBI->biYPelsPerMeter = pcxHdr.wYResolution;
lpBI->biClrUsed = 0;
lpBI->biClrImportant = 0;

// 分配内存以读取编码后的像素
lpSrc = new BYTE[file.GetLength() - sizeof(PCXHEADER) - 769];
lpTemp = lpSrc;

// 读取编码后的像素
if (file.ReadHuge(lpSrc, file.GetLength() - sizeof(PCXHEADER) - 769) !=
    file.GetLength() - sizeof(PCXHEADER) - 769 )
{
    // 大小不对。

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) hDIB);

    // 释放内存
    ::GlobalFree((HGLOBAL) hDIB);

    // 返回NULL。
    return NULL;
}

// 计算DIB中像素位置
lpDst = (BYTE *) FindDIBBits(pDIB);

// 一行一行解码
for (j = 0; j <wHeight; j++)
{
    i = 0;
    while (i < wWidth)
    {
        // 读取一个字节
        bChar = *lpTemp;
        lpTemp++;

        if ((bChar & 0xC0) == 0xC0)
        {
            // 行程
```

```
            iCount = bChar & 0x3F;

            // 读取下一个字节
            bChar = *lpTemp;
            lpTemp++;

            // bChar重复iCount次保存
            memset(&lpDst[(wHeight - j - 1) * lLineBytes + i], bChar, iCount);

            // 已经读取像素的个数加iCount
            i += iCount;
        }
        else
        {
            // 保存当前字节
            lpDst[(wHeight - j - 1) * lLineBytes + i] = bChar;

            // 已经读取像素的个数加1
            i += 1;
        }
    }
}

// 释放内存
delete lpSrc;

//*************************************************************
// 调色板

// 读调色板标志位
file.Read(&bChar, 1);
if (bChar != 0x0C)
{
    // 出错

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) hDIB);

    // 释放内存
    ::GlobalFree((HGLOBAL) hDIB);

    // 返回NULL。
    return NULL;
}

// 分配内存以读取编码后的像素
lpSrc = new BYTE[768];

// 计算DIB中调色板的位置
lpDst = (BYTE *) pDIB + sizeof(BITMAPINFOHEADER);
```

```
    // 读取调色板
    if (file.Read(lpSrc, 768) != 768)
    {
        // 大小不对。

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) hDIB);

        // 释放内存
        ::GlobalFree((HGLOBAL) hDIB);

        // 返回NULL。
        return NULL;
    }

    // 给调色板赋值
    for (i = 0; i < 256; i++)
    {
        lpDst[i * 4] = lpSrc[i * 3 + 2];
        lpDst[i * 4 + 1] = lpSrc[i * 3 + 1];
        lpDst[i * 4 + 2] = lpSrc[i * 3];
        lpDst[i * 4 + 3] = 0;
    }

    // 释放内存
    delete lpSrc;

    // 解除锁定
    ::GlobalUnlock((HGLOBAL) hDIB);

    // 返回DIB句柄
    return hDIB;
}
```

下面的代码是图 11－6 中菜单“加载 PCX 文件”的处理事件。该菜单项主要功能是加载一个用户指定的 PCX 文件，调用 ReadPCX256()函数将 PCX 图像转换成 DIB 格式，并替换当前显示的图像。

```
void CCh1_1View::OnCodeIRLE()
{
    // 加载256色PCX文件

    // 文件路径
    CString strFilePath;

    // 创建Open对话框
    CFileDialog dlg(TRUE, "PCX", NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
        "PCX图像文件 (*.PCX)|*.PCX|所有文件 (*.*)|*.*||", NULL);

    // 提示用户选择保存的路径
    if (dlg.DoModal() != IDOK)
```

```
{
    // 返回
    return;
}

// 获取用户指定的文件路径
strFilePath = dlg.GetPathName();

// CFile和CFileException对象
CFile file;
CFileException fe;

// 尝试打开指定的PCX文件
if (!file.Open(strFilePath, CFile::modeRead | CFile::shareDenyWrite, &fe))
{
    // 提示用户
    MessageBox("打开指定PCX文件时失败! ", "系统提示" ,
        MB_ICONINFORMATION | MB_OK);

    // 返回
    return;
}

// 调用ReadPCX256()函数读取指定的256色PCX文件
HDIB hDIB = ::ReadPCX256(file);

if (hDIB != NULL)
{
    // 提示用户
    //MessageBox("成功读取PCX文件! ", "系统提示" ,
    //  MB_ICONINFORMATION | MB_OK);

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 替换DIB, 同时释放旧DIB对象
    pDoc->ReplaceHDIB(hDIB);

    // 更新DIB大小和调色板
    pDoc->InitDIBData();

    // 设置脏标记
    pDoc->SetModifiedFlag(TRUE);

    // 重新设置滚动视图大小
    SetScrollSizes(MM_TEXT, pDoc->GetDocSize());

    // 实现新的调色板
    OnDoRealize((WPARAM)m_hWnd,0);

    // 更新视图
```

```
        pDoc->UpdateAllViews(NULL);
    }
    else
    {
        // 提示用户
        MessageBox("读取PCX文件失败！", "系统提示",
            MB_ICONINFORMATION | MB_OK);
    }

}
```

运行上述代码，即可加载任何256色的PCX图像文件，并保存为BMP格式。

11.4 LZW编码

11.4.1 理论基础

LZW编码又称为字串表（String Table）编码，它与哈大曼编码有点类似。哈大曼编码的编码原则是将出现概率高的数据以字符位数较少的码来表示，而出现概率低的数据则以位数较多的码表示；LZW编码则是将原始数据中的重复字符串建立一个字串表，然后用该重复字串在字串表中的索引（相当于哈夫曼编码中的码字）来替代原始数据以达到压缩的目的。LZW编码的编码效率是相当高的，但是解码速度将受到影响。LZW编码是一种无损的编码方法。

LZW（Lempel－Ziv & Welch编码法）是Welch将Lempel和Ziv所提出的数据编码方法改良后产生的压缩技术，不过由于标准的LZW编码在实际应用中仍有缺点，因此从标准的LZW编码法中又衍生了许多编码方法。其中DOS中的ARC、PKZIP和UNIX中的COMPRESS等数据压缩软件，其压缩技术都是标准LZW压缩技术的延伸。通常使用的GIF图像文件格式也是由标准LZW编码改进而来。

下面给出GIF－LZW编码的编码过程。它的示意代码如下所示：

```
// 初始化字串表
InitializeStringTable();

// 输出LZW CLEAR
WriteCode(LZW_CLEAR);

// 初始化保留字串Ω为空
Ω = NULL;

// 对输入的每个字符循环操作
for each character in the input stream
{
    // 获取下一个字符串
    k = GetNextCharacter();

    // 判断Ω+ k是否在字串表中
    if Ω+ k is in string table
```

```
        {
            // 将Ω赋值为Ω+ k;
            Ω  =  Ω+ k;
        }
        else
        {
            //Ω~ k不在字串表中

            // 输出字符串Ω的编码
            WriteCode(CodeFromString(Ω));

            // 将Ω添加到字串表中
            AddTableEntry(Ω);

            // 将Ω赋值为k
            Ω  = k;
        }
    }

    // 输出字符串Ω的编码
    WriteCode(CodeFromString(Ω));

    // 输出结束标志
    WriteCode(LZW_EOI);
```

下面我们通过一个示例来解说上述的编码过程。假设有一个长度为 9 的输入字符串：BBBCCBBAA，其编码过程如下。

首先，字串表初始化为：

表 11—3　　　　初始字串表

索引	值
<0x000>	0x000
<0x001>	0x000
<0x002>	0x000
<0x003>	0x000
…	…
<0x041>	0x041 (A)
<0x042>	0x041 (B)
<0x043>	0x041 (C)
<0x044>	0x041 (D)
…	…
<0x0FE>	0x0FE
<0x0FF>	0x0FF
<0x100>	LZW_CLEAR
<0x101>	LZW_EOI

按照上述流程，首先应该输出 LZW_CLEAR 的编码（即 LZW_CLEAR 在字串表中的索引 0x100），并初始化保留字符串 Ω 为空。

接着读出下一个字符“B”。因为 Ω + B = B 已经存在于字串表中，因此本步将不输出任何编码，而只是将 Ω 赋值为“B”。

接着读入下一个字符“B”。因为字串表中不存在字符串 Ω + B = BB，因此输出 Ω=B 在字串表中的编码（索引）0x042，并将 Ω 赋值为新读入的字符“B”，同时现字串表尾部中添加一个字串“BB”，其索引为<0x102>。

接着读入第 3 个字符“B”。因为 Ω + B = BB 已经存在于字串表中（上一步添加的），因此本步将不输出任何编码，而只是将 Ω 赋值为“BB”。

接着读入第 4 个字符“C”。因为字串表中不存在字符串 Ω + C = BBC，因此输出 Ω =BB 在字串表中的编码（索引）0x102，并将 Ω 赋值为新读入的字符“C”，同时现字串表尾部中添加一个字串“BBC”，其索引为<0x103>。

接着读入第 5 个字符“C”。因为字串表中不存在字符串 Ω + C = CC，因此输出 Ω =C 在字串表中的编码（索引）0x043，并将 Ω 赋值为新读入的字符“C”，同时现字串表尾部中添加一个字串“CC”，其索引为<0x104>。

接着读入第 6 个字符“B”。因为字串表中不存在字符串 Ω + B = CB，因此输出 Ω =C 在字串表中的编码（索引）0x043，并将 Ω 赋值为新读入的字符“B”，同时现字串表尾部中添加一个字串“CB”，其索引为<0x105>。

接着读入第 7 个字符“B”。因为 Ω + B = BB 已经存在于字串表中，因此本步将不输出任何编码，而只是将 Ω 赋值为“BB”。

接着读入第 8 个字符“A”。因为字串表中不存在字符串 Ω + A = BBA，因此输出 Ω =BB 在字串表中的编码（索引）0x102，并将 Ω 赋值为新读入的字符“A”，同时现字串表尾部中添加一个字串“BBA”，其索引为<0x106>。

接着读入第 9 个字符“A”。因为字串表中不存在字符串 Ω + A = AA，因此输出 Ω =A 在字串表中的编码（索引）0x041，并将 Ω 赋值为新读入的字符“A”，同时现字串表尾部中添加一个字串“AA”，其索引为<0x107>。

现在输入字符串全部读完，直接输出输出 Ω =A 在字串表中的编码（索引）0x41。

然后再输出 LZW_EOI 的编码 0x101，编码完成。

表 11－4 列出来上述完整的编码过程。

表 11－4　　LZW 编码示例

输入数据	输出结果	保留字符串值	添加的字串及其在字串表中的索引
	<0x100>	Ω =NULL	
B		Ω =B	
B	<0x042>	Ω =B	<0x102>　BB
B		Ω =BB	
C	<0x102>	Ω =C	<0x103>　BBC
C	<0x043>	Ω =C	<0x104>　CC
B	<0x043>	Ω =B	<0x105>　CB
B		Ω =BB	

续表

输入数据	输出结果	保留字符串值	添加的字串及其在字串表中的索引
A	<0x102>	Ω=A	<0x106> BBA
A	<0x041>	Ω=A	<0x107> AA
	<0x041>		
	<0x101>		

GIF－LZW 解码过程比较复杂，它和编码过程正好相反，其原则是将所有编码后的码字转换成对应的字符串，重新生成字串表，然后依次输出对应的字符串即可。

下面给出 GIF－LZW 编码的解码过程。它的示意代码所下所示：

```
// 依次处理每个编码
while((Code = GetNextCode()) != LZW_EOI)
{
    // 判断是否是LZW_CLEAR
    if (Code == LZW_CLEAR);
    {
        // 初始化字串表
        InitializeStringTable();

        // 获取下一个编码
        Code = GetNextCode();

        // 判断是否是LZW_CLEAR
        if (Code != LZW_CLEAR);
        {
            // 输出编码对应的字符串
            WriteString(StringFromCode(Code));

            // 更新OldCode
            OldCode = Code;
        }
    }
    else
    {
        // 判断字串表中是否存在该Code
        if (IsInTable(Code))
        {
            // 在字串表中

            // 输出编码对应的字符串
            WriteString(StringFromCode(Code));

            // 添加字符串入字串表
            AddStringToTable(StringFromCode(OldCode) +
                GetFirstChar(StringFromCode(Code)));

            // 更新OldCode
            OldCode = Code;
```

```
            }
            else
            {
                // 不在字串表中

                // 计算输出字符串
                OutString = StringFromCode(OldCode) +
                    GetFirstChar(StringFromCode(OldCode));

                // 输出字符串
                WriteString(OutString);

                // 添加字符串入字串表
                AddStringToTable(OutString);

                // 更新OldCode
                OldCode = Code;
            }
        }
    }
```

下面我们仍然通过示例来解说上述的解码过程。上面编码中我们已经将字符串BBBCCBBAA编码为0x100 0x042 0x102 0x043 0x043 0x102 0x041 0x041 0x101。现在我们对它进行解码。

首先读取第一个编码Code = 0x100，由于该Code为LZW_CLEAR，因此要对字串表进行初始化。初始化结果如表11－3所示。接着再读取下一个编码Code = 0x042，它不等于LZW_CLEAR，因此输出该编码对应的字符串“B”，同时更新OldCode为Code（OldCode = 0x042）。

读取下一个（第3个）编码Code = 0x102，由于该编码对应的字符串还没有生成（IsInTable(Code)返回False），需要先生成该字符串OutString：等于OldCode对应的字符串（B）加上该字符串的第一个字符，因此OutString = BB。输出该字符串，同时将该字符串BB添加到字串表中（其索引为<0x102>），更新OldCode为Code（OldCode = 0x102）。

读取下一个（第4个）编码Code = 0x043，它不等于LZW_CLEAR，而且字串表存在该索引，因此输出该编码对应的字符串“C”，然后将OldCode对应的字符串（BB）加上该字符串第一个字符（C）添加到字串表中（字符串BBC，其索引为<0x103>），同时更新OldCode为Code（OldCode = 0x043）。

读取下一个（第5个）编码Code = 0x043，它不等于LZW_CLEAR，而且字串表存在该索引，因此输出该编码对应的字符串“C”，然后将OldCode对应的字符串（C）加上该字符串第一个字符（C）添加到字串表中（字符串CC，其索引为<0x104>），同时更新OldCode为Code（OldCode = 0x043）。

读取下一个（第6个）编码Code = 0x102，它不等于LZW_CLEAR，而且字串表存在该索引，因此输出该编码对应的字符串“BB”，然后将OldCode对应的字符串（C）加上该字符串第一个字符（B）添加到字串表中（字符串CB，其索引为<0x105>），同时更新OldCode为Code（OldCode = 0x102）。

读取下一个（第7个）编码Code = 0x041，它不等于LZW_CLEAR，而且字串表存在该

索引，因此输出该编码对应的字符串“A”，然后将 OldCode 对应的字符串（BB）加上该字符串第一个字符（A）添加到字串表中（字符串 BBA，其索引为<0x106>），同时更新 OldCode 为 Code（OldCode = 0x041）。

读取下一个（第 8 个）编码 Code = 0x041，它不等于 LZW_CLEAR，而且字串表存在该索引，因此输出该编码对应的字符串“A”，然后将 OldCode 对应的字符串（A）加上该字符串第一个字符（A）添加到字串表中（字符串 AA，其索引为<0x107>），同时更新 OldCode 为 Code（OldCode = 0x041）。

读取最后一个（第 9 个）编码 Code = 0x101，它等于 LZW_EOI，表明数据已经完全解码，结束循环。

完整的解码过程如表 11－5 所示。

表 11－5　　　　LZW 解码示例

输入数据	输出结果	保留编码值	添加的字串及其在字串表中的索引
<0x100>			
<0x042>	B	OldCode = 0x041	
<0x102>	BB	OldCode = 0x102	<0x102>　BB
<0x043>	C	OldCode = 0x043	<0x103>　BBC
<0x043>	C	OldCode = 0x043	<0x104>　CC
<0x102>	BB	OldCode = 0x102	<0x105>　CB
<0x041>	A	OldCode = 0x041	<0x106>　BBA
<0x041>	A	OldCode = 0x041	<0x107>　AA
<0x101>			

至此，GIF－LZW 的编码和解码的流程已经介绍完毕。但是在真正编写一个完善高效的编码解码程序时，还要考虑到一些实际的问题。比如说字串表的大小问题（通过上面的示例可以发现，字串表增长的速度是很快的）和字串表的搜索问题等等。

11.4.2　GIF 文件格式

GIF（Graphics Interchange Format）是由美国 CompuServe 公司（它拥有全美 BBS 商业网络——CompuServe）在 1987 年所提出的图像文件格式，其最初目的是希望每个 BBS（电子公告板系统，Bulletin Board System）的使用者能够通过 GIF 图像文件轻易存储并交换图像数据，这也就是该图像格式被命名为“图像交换格式”的原因。

GIF 图像文件格式目前有两个版本，它们分别是 1987 年公布的 GIF87a 和 1989 年公布的 GIF89a。本节中将重点介绍 GIF89a 版本。

虽然 GIF 支持的图像色彩最多仅到 256 色，但是由于它具有极佳的压缩效率而早已被广泛接纳采用。

GIF 图像文件采用的是一种改良的 LZW 压缩算法，通常称为 GIF－LZW 压缩算法。下一小节将详细介绍该压缩算法与标准 LZW 压缩算法的异同。

GIF 图像文件是以块（又称为区域结构）的方式来存储图像相关的信息，常用的 GIF 图像文件块如表 11－6 所示。

表 11－6 GIF 常用的块

块名称	英文描述
文件头信息	header
逻辑屏幕描述块	Logical Screen Descriptor
全局调色板	Global Color Table
图像描述块	Image Descriptor
局部调色板	Local Color Table
图像压缩数据	Table Based Image Data
图像控制扩充块	Graphic Control Extension
图像说明扩充块	Plain Text Extension
图像注释扩充块	Comment Extension
应用程序扩充块	Application Extension
文件结尾块	Trailer

GIF89a 按照这些块的特征将上述块分为三大类：

➢ 控制块（Control Block）

控制块包含了控制数据流的处理以及硬件参数的设置，其成员包括文件头信息、逻辑屏幕描述块、图像控制扩充块和文件结尾块。

➢ 图像描述块（Graphic Rendering Block）

图像描述块包含了在显示设备上描述图像所需的信息，其成员包括图像描述块、全局调色板、局部调色板、图像压缩数据和图像说明扩充块。

➢ 特殊用途块（Special Purpose Block）

特殊用途块包含了与图像数据处理无直接关系的信息，其成员包括图像注释扩充块和应用程序扩充块。

下面详细介绍一下各个块的结构。

1. 文件头信息

GIF 的文件头只有 6 个字节，其结构定义如下：

```
typedef struct gifheader
   {
      BYTE bySignature[3];
      BYTE byVersion[3];
   } GIFHEADER;
```

其中 bySignature 为 GIF 文件标识码，其值固定为“GIF”（不含 NULL），使用者可以通过该域来判断一个图像文件是否是 GIF 图像格式文件。

ByVersion 表明 GIF 文件的版本信息。其取值固定为“87a”或者“89a”（不含 NULL）。分别表示 GIF 文件的版本为 GIF87a 版或 GIF89a 版。

2. 逻辑屏幕描述块

逻辑屏幕（Logical Screen）是一个虚拟屏幕（Visual Screen）。通过逻辑屏幕可以知道如何显示图像。逻辑屏幕描述块结构定义如下：

```
typedef struct gifscrdesc
  {
    WORD wWidth;
    WORD wDepth;
    struct globalflag
      {
        BYTE PalBits   : 3;
        BYTE SortFlag  : 1;
        BYTE ColorRes  : 3;
        BYTE GlobalPal : 1;
      } GlobalFlag;
    BYTE byBackground;
    BYTE byAspect;
  } GIFSCRDESC;
```

其中：

- wWidth 用来指定逻辑屏幕的宽度；
- wDepth 用来指定逻辑屏幕的高度；
- GlobalFlag 为全域性数据，它的总长度为一个字节（包含 8 位），其中的前 3 位（第 0 到第 2 位，GlobalFlag.PalBits）指定全局调色板的位数，可以根据该值来计算全局调色板的大小：

  $$全局调色板的大小 = 3\times 2^{(GlobalFlag.PalBits+1)}$$

  第 3 位（GlobalFlag.SortFlag）指明全局调色板中的 RGB 颜色值是否经过排序。其值为 1 表示调色板中的 RGB 颜色值是按照其使用率（即颜色的重要性）进行从高到低的次序排序的。第 4 到第 6 位（GlobalFlag.ColorRes）指定图像的色彩分辨率（Color Resolution），其计算方法为：

  $$图像的色彩分辨率 = 3\times 2^{(GlobalFlag.ColorRes+1)}$$

  第 7 位（GlobalFlag.GlobalPal）指明 GIF 文件中是否有全局调色板。其取值为 1 表示有全局调色板。
- byBackground 用来指定逻辑屏幕的背景颜色。当图像小于逻辑屏幕时，未被图像覆盖的区域的颜色由该值指定。
- byAspect 用来指定逻辑屏幕的像素长宽比例。

3. 全局调色板

全局调色板的大小由 GlobalFlag.PalBits 来决定，其最大长度为 768（3×256）字节。全局调色板的数据（RGB）是按照 RGBRGB……RGB 的方式存储的。

4. 图像描述块

一个 GIF 图像文件中可以存储多幅图像（图像描述块＋图像扩充块），而且这些图像没有固定的存放次序。因此 GIF 用一个字节的识别码（Image Separator）来判断接下来的数据是否是图像描述块，其值为 0x2C。

图像描述块的结构定义如下：

```
typedef struct gifimage
    {
        WORD wLeft;
        WORD wTop;
        WORD wWidth;
        WORD wDepth;
        struct localflag
            {
                BYTE PalBits    : 3;
                BYTE Reserved   : 2;
                BYTE SortFlag   : 1;
                BYTE Interlace  : 1;
                BYTE LocalPal   : 1;
            } LocalFlag;
    } GIFIMAGE;
```

其中：

- wLeft 用来指定图像相对逻辑屏幕左上角的 X 坐标（以像素为单位）；
- wTop 用来指定图像相对逻辑屏幕左上角的 Y 坐标（以像素为单位）；
- wWidth 用来指定图像的宽度（以像素为单位）；
- wDepth 用来指定图像的高度（以像素为单位）；
- LocalFlag 用来指定区域性数据，它的总长度为一个字节（包含 8 位），其中的前 3 位（第 0 到第 2 位，LocalFlag.PalBits）指定局部调色板的位数，可以根据该值来计算局部调色板的大小：

$$局部调色板的大小 = 3\times 2^{(LocalFlag.PalBits+1)}$$

第 3－4 位（LocalFlag.Reserved）为保留位，未用，其取值固定为 0。第 5 位（LocalFlag.SortFlag）指明局部调色板中的 RGB 颜色值是否经过排序，其值为 1 表示调色板中的 RGB 颜色值是按照其使用率（即颜色的重要性）进行从高到低的次序排序的。第 6 位（LocalFlag. Interlace）指明 GIF 图像是否以交错方式存储，其取值为 1 表示以交错方式存储。以交错方式存储的图像数据的好处是无需将整个图像文件解压完成就可以看到图像的概貌（Rough View），这对于原本期望成为网络上图像数据交换标准的 GIF 图像文件来说的确是一个极佳的设计。当图像是按照交错方式存储时，其图像数据的处理可以分为 4 个阶段（Pass）：第一阶段从第 0 行开始，每次间隔 8 行进行处理；第二阶段从第 4 行开始，每次间隔 8 行进行处理；第三阶段从第 2 行开始，每次间隔 4 行进行处理；第四阶段从第 1 行开始，每次间隔 2 行进行处理。这样当完成第一阶段时就可以看到图像的概貌，当处理完第二阶段时图像将变得清晰一些；当处理完第三阶段时，图像处理完成一半，清晰效果也进一步增强；当完成第四阶段，图像完全处理完毕。第 7 位（LocalFlag. LocalPal）指明 GIF 图像是否包含局部调色板。其取值为 1 表示有局部调色板。

5. 局部调色板

局部调色板的大小由 LocalFlag.PalBits 来决定，其最大长度为 768（3×256）字节。原则上 GIF 图像文件中每张图像都有其专属的局部调色板，如果某个 GIF 文件中没有指定局部调

色板，则应该用全局调色板来替代。局部调色板的数据（RGB）也是按照RGBRGB……RGB的方式存储的。

6. 图像压缩数据

图像数据是按照GIF－LZW压缩编码后存储于图像压缩数据块中的。由于GIF－LZW压缩编码的需要，必须首先存储GIF－LZW的最小编码长度（LZW Minium Code Size）以供解码程序使用，然后再存储编码后的图像数据。编码后的图像数据是以一个个数据子块的方式存储的，每个数据子块的最大长度为256字节。数据子块的第一个字节指定该数据子块的长度，接下来的数据为数据子块中的内容。如果某个数据子块的第一个字节数值为0，即该数据子块中没有包含任何有用数据，则该子块称为块终结符（Block Terminator），用来标识数据子块到此结束。

图像压缩数据的结构如图11－8所示。

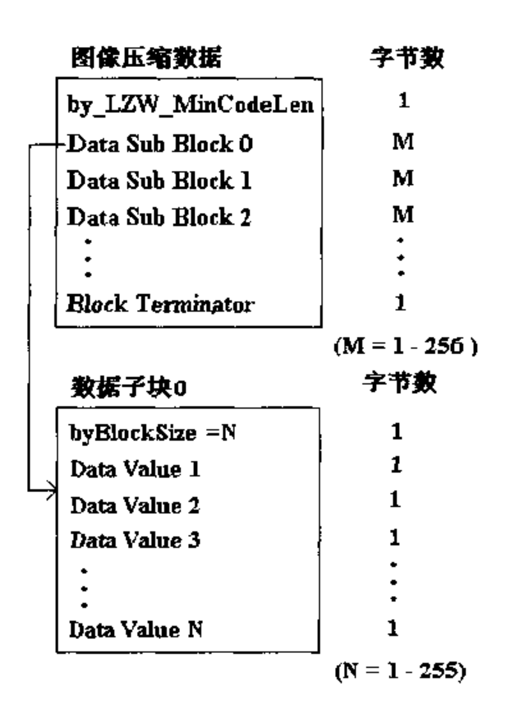


图11－8 GIF图像压缩数据结构示意图

7. 图像控制扩充块

GIF图像文件中可以包含多个扩充块，而且各个扩充块是按照任意次序放置的。因此GIF用一个字节的识别码来标识该块是否是扩充块，扩充块的识别码数值为0x21。

图像控制扩充块描述了与图像控制相关的参数，GIF中用识别码0xF9来判断一个扩充块是否为图像控制扩充块。它的结构定义如下：

```
typedef struct gifcontrol
  {
    BYTE byBlockSize;
    struct flag
```

```
        {
            BYTE Transparency    : 1;
            BYTE UserInput       : 1;
            BYTE DisposalMethod  : 3;
            BYTE Reserved        : 3;
        } Flag;
    WORD wDelayTime;
    BYTE byTransparencyIndex;
    BYTE byTerminator;
} GIFCONTROL;
```

其中：

- byBlockSize 用来指定该图像控制扩充块的长度，其取值固定为 6；
- Flag 用来描述图像控制相关数据，它的长度为 1 字节。它的第 0 位（Flag.Transparency）用来指定图像中是否有具有透明性的颜色。如果该位为 1，这表明图像某种颜色具有透明性，该颜色由参数 byTransparencyIndex 指定。第 1 位（Flag.UserInput）用来判断在显示一幅图像后，是否需要用户输入后再进行下一个动作。如果该位为 1，表示应用程序在进行下一个动作之前需要用户输入（可能是键盘输入，也可能是鼠标输入，这由应用程序自行决定）。第 2－4 位（Flag.DisposalMethod）用来指定图像显示后的处理方式，当该值为 0 时，表示没有指定任何处理方式；当该值为 1 时，表明不进行任何处理动作；当该值为 2 时，表明图像显示后以背景色擦去；当该值为 3 时，表明图像显示后恢复原先的背景图像。第 5－7 位（Flag.Reserved）为保留位，没有任何含义，固定为 0
- wDelayTime 用来指定应用程序进行下一步操作前延迟的时间（以毫秒为单位）。如果 Flag.UserInput 和 wDelayTime 都设定了，则以先发者为主：如果没有到指定的延迟时间即有用户输入，则应用程序直接进行下一步操作；如果到达延迟时间后还没有用户输入，应用程序也直接进行下一步操作。
- byTransparencyIndex 用来指定图像中透明色的颜色索引。指定的透明色将不在显示设备上显示。
- byTerminator 为块终结符，其值固定为 0。

8. 图像说明扩充块

图像说明扩充块中定义了与图像同时显示的文字信息。GIF 中用识别码 0x01 来判断一个扩充块是否为图像说明扩充块。它的结构定义如下：

```
typedef struct gifplaintext
    {
        BYTE byBlockSize;
        WORD wTextGridLeft;
        WORD wTextGridTop;
        WORD wTextGridWidth;
        WORD wTextGridDepth;
        BYTE byCharCellWidth;
        BYTE byCharCellDepth;
        BYTE byForeColorIndex;
        BYTE byBackColorIndex;
```

```
    } GIFPLAINTEXT;
```

其中：

- byBlockSize 用来指定该图像扩充块的长度，其取值固定为 13；
- wTextGridLeft 用来指定文字显示方格相对于逻辑屏幕左上角的 X 坐标（以像素为单位）。
- wTextGridTop 用来指定文字显示方格相对于逻辑屏幕左上角的 Y 坐标（以像素为单位）。
- wTextGridWidth 用来指定文字显示方格的宽度（以像素为单位）。
- wTextGridDepth 用来指定文字显示方格的高度（以像素为单位）。
- byCharCellWidth 用来指定字符的宽度（以像素为单位）。
- byCharCellDepth 用来指定字符的高度（以像素为单位）。
- byForeColorIndex 用来指定字符的前景色（颜色索引值）。
- byBackColorIndex 用来指定字符的背景色（颜色索引值）。

紧接着图像说明扩充块后就是图像说明的数据子块（Plain Text Data Sub Block）。它的存储方式就是按照前面介绍的数据子块的存储方式：第 0 个字节存储数据子块的大小，接下来再存储数据内容。其中最后一个数据子块应该为 1 字节的块终结符，其值固定为 0。

9. 图像注释扩充块

图像注释扩充块包含了图像的文字注释说明。GIF 中用识别码 0xFE 来判断一个扩充块是否为图像注释扩充块。图像注释扩充块中的数据子块个数不限，必须通过块终结符来判断该扩充块是否结束。

10. 应用程序扩充块

应用程序扩充块包含了制作该 GIF 图像文件的应用程序信息。GIF 中用识别码 0xFF 来判断一个扩充块是否为应用程序扩充块。它的结构定义如下：

```
typedef struct gifapplication
    {
        BYTE byBlockSize;
        BYTE byIdentifier[8];
        BYTE byAuthentication[3];
    } GIFAPPLICATION;
```

其中：

- byBlockSize 用来指定该应用程序扩充块的长度，其取值固定为 12；
- byIdentifie 用来指定应用程序名称。
- byAuthentication 用来指定应用程序的识别码。

11. 文件结尾块

文件结尾块为 GIF 图像文件最后一个字节，其取值固定为 0x3B。

11.4.3 编程实现 GIF 文件的读写

知道了 GIF 文件的格式和 GIF－LZW 编码的原则，就可以编程实现读写 GIF 图像文件的功能。首先可以定义 GIFAPI 函数库来实现 GIF 图像文件的读写操作。下面给出该 API 函

数库的源代码。

1. GIFAPI.h 源代码

```
//////////////////////////////////////////////////////////////
// GIFAPI.h


// GIF的文件头的结构定义
typedef struct gifheader
    {
      BYTE bySignature[3];
      BYTE byVersion[3];
    } GIFHEADER;

// 逻辑屏幕描述块的结构定义
typedef struct gifscrdesc
    {
      WORD wWidth;
      WORD wDepth;
      struct globalflag
         {
           BYTE PalBits   : 3;
           BYTE SortFlag  : 1;
           BYTE ColorRes  : 3;
           BYTE GlobalPal : 1;
         } GlobalFlag;
      BYTE byBackground;
      BYTE byAspect;
    } GIFSCRDESC;

// 图像描述块的结构定义
typedef struct gifimage
    {
      WORD wLeft;
      WORD wTop;
      WORD wWidth;
      WORD wDepth;
      struct localflag
         {
           BYTE PalBits   : 3;
           BYTE Reserved  : 2;
           BYTE SortFlag  : 1;
           BYTE Interlace : 1;
           BYTE LocalPal  : 1;
         } LocalFlag;
    } GIFIMAGE;

// 图像控制扩充块的结构定义
typedef struct gifcontrol
    {
      BYTE byBlockSize;
```

```
    struct flag
        {
            BYTE Transparency   : 1;
            BYTE UserInput      : 1;
            BYTE DisposalMethod : 3;
            BYTE Reserved       : 3;
        } Flag;
    WORD wDelayTime;
    BYTE byTransparencyIndex;
    BYTE byTerminator;
  } GIFCONTROL;

// 图像说明扩充块的结构定义
typedef struct gifplaintext
  {
    BYTE byBlockSize;
    WORD wTextGridLeft;
    WORD wTextGridTop;
    WORD wTextGridWidth;
    WORD wTextGridDepth;
    BYTE byCharCellWidth;
    BYTE byCharCellDepth;
    BYTE byForeColorIndex;
    BYTE byBackColorIndex;
  } GIFPLAINTEXT;

// 应用程序扩充块的结构定义
typedef struct gifapplication
  {
    BYTE byBlockSize;
    BYTE byIdentifier[8];
    BYTE byAuthentication[3];
  } GIFAPPLICATION;

typedef struct gifd_var
  {
    LPSTR lpDataBuff;
    LPSTR lpBgnBuff;
    LPSTR lpEndBuff;
    DWORD dwDataLen;
    WORD  wMemLen;
    WORD  wWidth;
    WORD  wDepth;
    WORD  wLineBytes;
    WORD  wBits;
    BOOL  bEOF;
    BOOL  bInterlace;
  } GIFD_VAR;
typedef GIFD_VAR FAR *LPGIFD_VAR;

typedef struct gifc_var
```

```
    LPSTR lpDataBuff;
    LPSTR lpEndBuff;
    DWORD dwTempCode;
    WORD  wWidth;
    WORD  wDepth;
    WORD  wLineBytes;
    WORD  wBits;
    WORD  wByteCnt;
    WORD  wBlockNdx;
    BYTE  byLeftBits;
} GIFC_VAR;
typedef GIFC_VAR FAR *LPGIFC_VAR;


// 宏运算
#define DWORD_WBYTES(x)         ( (((x) + 31UL) >> 5) << 2 )
#define WORD_WBYTES(x)          ( (((x) + 15UL) >> 4) << 1 )
#define BYTE_WBYTES(x)          (  ((x) +  7UL) >> 3       )

//常量
#define MAX_BUFF_SIZE           32768 /* 32K */
#define MAX_HASH_SIZE            5051
#define MAX_TABLE_SIZE           4096 /* 12 bit */
#define MAX_SUBBLOCK_SIZE         255

// 函数原型
BOOL WINAPI DIBToGIF(LPSTR lpDIB, CFile& file, BOOL bInterlace);
void WINAPI EncodeGIF_LZW(LPSTR lpDIBBits, CFile& file,
                          LPGIFC_VAR lpGIFCVar,WORD wWidthBytes, BOOL bInterlace);
void WINAPI GIF_LZW_WriteCode(CFile& file, WORD wCode, LPSTR lpSubBlock,
                              LPBYTE lpbyCurrentBits,LPGIFC_VAR lpGIFCVar);
HDIB WINAPI ReadGIF(CFile& file);

void WINAPI ReadSrcData(CFile& file, LPWORD lpwMemLen, LPDWORD lpdwDataLen,
                        LPSTR lpSrcBuff, LPBOOL lpbEOF);
void WINAPI DecodeGIF_LZW(CFile& file, LPSTR lpDIBBits,
                          LPGIFD_VAR lpGIFDVar,WORD wWidthBytes);
```

2. GIFAPI.cpp 源代码

```
// ************************************************************************
//  文件名：GIFAPI.cpp
//
//  GIF(Graphics Interchange Format) API函数库：
//
//  DIBToGIF()            - 将指定的DIB对象（< 256色）保存为GIF文件
//  EncodeGIF_LZW()       - 对指定图像进行GIF_LZW编码
//  GIF_LZW_WriteCode()   - 输出一个编码
//  ReadGIF()             - 读取GIF文件
//  DecodeGIF_LZW()       - 对GIF_LZW编码结果进行解码
//  ReadSrcData()         - 读取GIF_LZW编码
```

```
//
// ************************************************************************

#include "stdafx.h"
#include "DIBAPI.h"
#include "GIFAPI.h"

#include <io.h>
#include <errno.h>

#include <math.h>
#include <direct.h>

/*************************************************************************
 *
 * 函数名称:
 *   DIBToGIF()
 *
 * 参数:
 *   LPSTR lpDIB          - 指向DIB对象的指针
 *   CFile& file          - 要保存的文件
 *   BOOL   bInterlace    - 是否按照交错方式保存
 *
 * 返回值:
 *   BOOL                 - 成功返回True，否则返回False。
 *
 * 说明:
 *   该函数将指定的DIB对象（< 256色）保存为GIF文件。
 *
 *************************************************************************/
BOOL WINAPI DIBToGIF(LPSTR lpDIB, CFile& file, BOOL bInterlace)
{
    // 循环变量
    WORD    i;
    WORD    j;

    // DIB高度
    WORD    wHeight;

    // DIB宽度
    WORD    wWidth;

    // 指向DIB像素指针
    LPSTR   lpDIBBits;

    // GIF文件头
    GIFHEADER          GIFH;

    // GIF逻辑屏幕描述块
    GIFSCRDESC         GIFS;
```

```
// GIF图像描述块
GIFIMAGE            GIFI;

// GIF编码参数
GIFC_VAR            GIFCVar;

// 颜色数目
WORD                wColors;

// 每行字节数
WORD                wWidthBytes;

// 调色板
BYTE                byGIF_Pal[768];

// 字节变量
BYTE                byChar;

// 指向BITMAPINFO结构的指针（Win3.0）
LPBITMAPINFO        lpbmi;

// 指向BITMAPCOREINFO结构的指针
LPBITMAPCOREINFO    lpbmc;

// 表明是否是Win3.0 DIB的标记
BOOL                bWinStyleDIB;

// 获取DIB高度
wHeight = (WORD) DIBHeight(lpDIB);

// 获取DIB宽度
wWidth  = (WORD) DIBWidth(lpDIB);

// 找到DIB图像像素起始位置
lpDIBBits = FindDIBBits(lpDIB);

// 给GIFCVar结构赋值
GIFCVar.wWidth      = wWidth;
GIFCVar.wDepth      = wHeight;
GIFCVar.wBits       = DIBBitCount(lpDIB);
GIFCVar.wLineBytes = (WORD)BYTE_WBYTES((DWORD)GIFCVar.wWidth *
                              (DWORD)GIFCVar.wBits);

// 计算每行字节数
wWidthBytes = (WORD)DWORD_WBYTES(wWidth * (DWORD)GIFCVar.wBits);

// 计算颜色数目
wColors     = 1 << GIFCVar.wBits;

// 获取指向BITMAPINFO结构的指针（Win3.0）
lpbmi = (LPBITMAPINFO)lpDIB;
```

```
// 获取指向BITMAPCOREINFO结构的指针
lpbmc = (LPBITMAPCOREINFO)lpDIB;

// 判断是否是WIN3.0的DIB
bWinStyleDIB = IS_WIN30_DIB(lpDIB);

// 给调色板赋值
if (bWinStyleDIB)
{
    j = 0;
    for (i = 0; i < wColors; i++)
    {
        // 读取红色分量
        byGIF_Pal[j++] = lpbmi->bmiColors[i].rgbRed;

        // 读取绿色分量
        byGIF_Pal[j++] = lpbmi->bmiColors[i].rgbGreen;

        // 读取蓝色分量
        byGIF_Pal[j++] = lpbmi->bmiColors[i].rgbBlue;
    }
}
else
{
    j = 0;
    for (i = 0; i < wColors; i++)
    {
        // 读取红色分量
        byGIF_Pal[j++] = lpbmc->bmciColors[i].rgbtRed;

        // 读取绿色分量
        byGIF_Pal[j++] = lpbmc->bmciColors[i].rgbtGreen;

        // 读取红色分量
        byGIF_Pal[j++] = lpbmc->bmciColors[i].rgbtBlue;
    }
}

/////////////////////////////////////////////////////////////////////////////////////////
// 开始写GIF文件

// 写GIF文件头
GIFH.bySignature[0] = 'G';
GIFH.bySignature[1] = 'I';
GIFH.bySignature[2] = 'F';
GIFH.byVersion[0]='8';
GIFH.byVersion[1]='9';
GIFH.byVersion[2]='a';
file.Write((LPSTR)&GIFH, 6);
```

```
// 写GIF逻辑屏幕描述块
GIFS.wWidth                  = GIFCVar.wWidth;
GIFS.wDepth                  = GIFCVar.wDepth;
GIFS.GlobalFlag.PalBits      = (BYTE)(GIFCVar.wBits - 1);
GIFS.GlobalFlag.SortFlag     = 0x00;
GIFS.GlobalFlag.ColorRes     = (BYTE)(GIFCVar.wBits - 1);
GIFS.GlobalFlag.GlobalPal    = 0x01;
GIFS.byBackground            = 0x00;
GIFS.byAspect                = 0x00;
file.Write((LPSTR)&GIFS, 7);

// 写GIF全局调色板
file.Write((LPSTR)byGIF_Pal, (wColors*3));

// 写GIF图像描述间隔符
byChar       = 0x2C;
file.Write((LPSTR)&byChar, 1);

// 写GIF图像描述块
GIFI.wLeft                   = 0;
GIFI.wTop                    = 0;
GIFI.wWidth                  = GIFCVar.wWidth;
GIFI.wDepth                  = GIFCVar.wDepth;
GIFI.LocalFlag.PalBits       = 0x00;
GIFI.LocalFlag.Reserved      = 0x00;
GIFI.LocalFlag.SortFlag      = 0x00;
GIFI.LocalFlag.Interlace     = (BYTE)(bInterlace ? 0x01 : 0x00);
GIFI.LocalFlag.LocalPal      = 0x00;
file.Write((LPSTR)&GIFI, 9);

// 写GIF图像压缩数据
HANDLE hSrcBuff = GlobalAlloc(GHND, (DWORD)MAX_BUFF_SIZE);
GIFCVar.lpDataBuff = (LPSTR)GlobalLock(hSrcBuff);
GIFCVar.lpEndBuff  = GIFCVar.lpDataBuff;
GIFCVar.dwTempCode = 0UL;
GIFCVar.wByteCnt   = 0;
GIFCVar.wBlockNdx  = 1;
GIFCVar.byLeftBits = 0x00;

// 进行GIF_LZW编码
EncodeGIF_LZW(lpDIBBits, file, &GIFCVar,wWidthBytes, bInterlace);

// 判断是否编码成功
if (GIFCVar.wByteCnt)
{
    // 写入文件
    file.Write(GIFCVar.lpDataBuff, GIFCVar.wByteCnt);
}

// 释放内存
GlobalUnlock(hSrcBuff);
```

```
    GlobalFree(hSrcBuff);

    // 写GIF Block Terminator
    byChar    = 0x00;
    file.Write((LPSTR)&byChar,1);

    // 写GIF文件结尾块
    byChar    = 0x3B;
    file.Write((LPSTR)&byChar,1);

    // 返回
    return TRUE;
}

/*************************************************************************
 *
 * 函数名称:
 *   EncodeGIF_LZW()
 *
 * 参数:
 *   LPSTR lpDIBBits         - 指向原DIB图像指针
 *   CFile& file             - 要保存的文件
 *   LPGIFC_VAR lpGIFCVar    - 指向GIFC_VAR结构的指针
 *   WORD wWidthBytes        - 每行图像字节数
 *   BOOL bInterlace         - 是否按照交错方式保存
 *
 * 返回值:
 *   无
 *
 * 说明:
 *   该函数对指定图像进行GIF_LZW编码。
 *
 *************************************************************************/
void WINAPI EncodeGIF_LZW(LPSTR lpDIBBits, CFile& file,
                          LPGIFC_VAR lpGIFCVar,WORD wWidthBytes, BOOL bInterlace)
{
    // 内存分配句柄
    HANDLE hTableNdx;
    HANDLE hPrefix;
    HANDLE hSuffix;

    // 指向字串表指针
    LPWORD lpwTableNdx;

    // 用于字串表搜索的索引
    LPWORD lpwPrefix;
    LPBYTE lpbySuffix;

    // 指向当前编码像素的指针
    LPSTR  lpImage;
```

```
// 计算当前数据图像的偏移量
DWORD  dwDataNdx;

// LZW_CLEAR
WORD    wLZW_CLEAR;

// LZW_EOI
WORD    wLZW_EOI;

// LZW_MinCodeLen
BYTE    byLZW_MinCodeLen;

// 字串表索引
WORD    wPreTableNdx;
WORD    wNowTableNdx;
WORD    wTopTableNdx;

// 哈希表索引
WORD    wHashNdx;
WORD    wHashGap;
WORD    wPrefix;
WORD    wShiftBits;

// 当前图像的行数
WORD    wRowNum;

WORD    wWidthCnt;

// 循环变量
WORD    wi;
WORD    wj;

// 交错方式存储时每次增加的行数
WORD    wIncTable[5]  = { 8,8,4,2,0 };

// 交错方式存储时起始行数
WORD    wBgnTable[5]  = { 0,4,2,1,0 };

BOOL    bStart;
BYTE    bySuffix;
BYTE    bySubBlock[256];
BYTE    byCurrentBits;
BYTE    byMask;
BYTE    byChar;
BYTE    byPass;

// 临时字节变量
BYTE    byTemp;

// 给字串表分配内存
hTableNdx         = GlobalAlloc(GHND,(DWORD)(MAX_HASH_SIZE<<1));
```

```
hPrefix          = GlobalAlloc(GHND, (DWORD)(MAX_HASH_SIZE<<1));
hSuffix          = GlobalAlloc(GHND, (DWORD)MAX_HASH_SIZE);

// 锁定内存
lpwTableNdx      = (LPWORD)GlobalLock(hTableNdx);
lpwPrefix        = (LPWORD)GlobalLock(hPrefix);
lpbySuffix       = (LPBYTE)GlobalLock(hSuffix);

// 计算LZW_MinCodeLen
byLZW_MinCodeLen = (BYTE)((lpGIFCVar->wBits>1) ? lpGIFCVar->wBits : 0x02);

// 写GIF LZW最小代码大小
file.Write((LPSTR)&byLZW_MinCodeLen,1);

wRowNum          = 0;
bStart           = TRUE;
byPass           = 0x00;

// 计算LZW_CLEAR
wLZW_CLEAR       = 1 << byLZW_MinCodeLen;

// 计算LZW_EOI
wLZW_EOI         = wLZW_CLEAR + 1;

// 初始化字串表
byCurrentBits    = byLZW_MinCodeLen + (BYTE)0x01;
wNowTableNdx     = wLZW_CLEAR + 2;
wTopTableNdx     = 1 << byCurrentBits;
for(wi=0; wi<MAX_HASH_SIZE; wi++)
{
    // 初始化为0xFFFF
    *(lpwTableNdx+wi) = 0xFFFF;
}

// 输出LZW_CLEAR
GIF_LZW_WriteCode(file, wLZW_CLEAR, (LPSTR)bySubBlock,
                  &byCurrentBits, lpGIFCVar);

// 逐行编码
for(wi=0; wi<lpGIFCVar->wDepth; wi++)
{
    // 计算当前偏移量
    dwDataNdx  = (DWORD)(lpGIFCVar->wDepth - 1 - wRowNum) * (DWORD)wWidthBytes;

    // 指向当前行图像的指针
    lpImage    = (LPSTR) (((BYTE*)lpDIBBits) + dwDataNdx);

    wWidthCnt  = 0;
    wShiftBits = 8 - lpGIFCVar->wBits;
    byMask     = (BYTE)((lpGIFCVar->wBits==1) ? 0x80 : 0xF0);
```

```
if (bStart)
{
    // 判断是否是256色位图（一个像素一字节）
    if (lpGIFCVar->wBits==8)
    {
        // 256色，直接赋值即可
        byTemp      = *lpImage++;
    }
    else
    {
        // 非256色，需要移位获取像素值
        wShiftBits  = 8 - lpGIFCVar->wBits;
        byMask      = (BYTE)((lpGIFCVar->wBits==1) ? 0x80 : 0xF0);
        byTemp      = (BYTE)((*lpImage & byMask) >> wShiftBits);
        byMask    >>= lpGIFCVar->wBits;
        wShiftBits -= lpGIFCVar->wBits;
    }
    wPrefix     = (WORD)byTemp;
    bStart      = FALSE;
    wWidthCnt ++;
}

// 每行编码
while(wWidthCnt < lpGIFCVar->wWidth)
{
    // 判断是否是256色位图（一个像素一字节）
    if (lpGIFCVar->wBits==8)
    {
        // 256色，直接赋值即可
        byTemp = *lpImage++;
    }
    else
    {
        // 非256色，需要移位获取像素值
        byChar = *lpImage;
        byTemp = (BYTE)((byChar & byMask) >> wShiftBits);
        if (wShiftBits)
        {
            byMask    >>= lpGIFCVar->wBits;
            wShiftBits -= lpGIFCVar->wBits;
        }
        else
        {
            wShiftBits  = 8 - lpGIFCVar->wBits;
            byMask      = (BYTE)((lpGIFCVar->wBits==1) ? 0x80 : 0xF0);
            lpImage    ++;
        }
    }
    bySuffix    = byTemp;
    wWidthCnt ++;
```

```
// 查找当前字符串是否存在于字串表中
wHashNdx = wPrefix ^ ((WORD)bySuffix << 4);
wHashGap = (wHashNdx ? (MAX_HASH_SIZE - wHashNdx) : 1);

// 判断当前字符串是否在字串表中
while(TRUE)
{
    // 当前字符串不在字串表中
    if (*(lpwTableNdx + wHashNdx) == 0xFFFF)
    {
       // 新字符串，退出循环
       break;
    }

    // 判断是否找到该字符串
    if ((*(lpwPrefix+wHashNdx)  == wPrefix) &&
        (*(lpbySuffix+wHashNdx) == bySuffix))
    {
        // 找到，退出循环
        break;
    }

    // 第二哈希表
    if (wHashNdx < wHashGap)
    {
        wHashNdx += MAX_HASH_SIZE;
    }
    wHashNdx -= wHashGap;
}

// 判断是否是新字符串
if (*(lpwTableNdx+wHashNdx) != 0xFFFF)
{
    // 不是新字符串
    wPrefix = *(lpwTableNdx + wHashNdx);
}
else
{
    // 新字符串

    // 输出该编码
    GIF_LZW_WriteCode(file,wPrefix,(LPSTR)bySubBlock,
                      &byCurrentBits,lpGIFCVar);

    // 将该新字符串添加到字串表中
    wPreTableNdx = wNowTableNdx;

    // 判断是否达到最大字串表大小
    if (wNowTableNdx < MAX_TABLE_SIZE)
    {
        *(lpwTableNdx+wHashNdx) = wNowTableNdx++;
```

```
            *(lpwPrefix+wHashNdx)   = wPrefix;
            *(lpbySuffix+wHashNdx)  = bySuffix;
        }

        if (wPreTableNdx == wTopTableNdx)
        {
            if (byCurrentBits<12)
            {
                byCurrentBits ++;
                wTopTableNdx <<= 1;
            }
            else
            {
                // 字串表到达最大长度

                // 输出LZW_CLEAR
                GIF_LZW_WriteCode(file, wLZW_CLEAR, (LPSTR)bySubBlock,
                        &byCurrentBits, lpGIFCVar);

                // 重新初始化字串表
                byCurrentBits     = byLZW_MinCodeLen + (BYTE)0x01;
                wLZW_CLEAR        = 1 << byLZW_MinCodeLen;
                wLZW_EOI          = wLZW_CLEAR + 1;
                wNowTableNdx      = wLZW_CLEAR + 2;
                wTopTableNdx      = 1 << byCurrentBits;
                for(wj=0;wj<MAX_HASH_SIZE;wj++)
                {
                    // 初始化为0xFFFF
                    *(lpwTableNdx+wj) = 0xFFFF;
                }
            }
        }
        wPrefix = (WORD)bySuffix;
    }
}

// 判断是否是交错方式
if (bInterlace)
{
    // 交错方式，计算下一行位置
    wRowNum += wIncTable[byPass];
    if (wRowNum>=lpGIFCVar->wDepth)
    {
        byPass ++;
        wRowNum = wBgnTable[byPass];
    }
}
else
{
    // 非交错方式，直接将行数加一即可
    wRowNum ++;
```

```
        }
    }

    // 输出当前编码
    GIF_LZW_WriteCode(file, wPrefix, (LPSTR)bySubBlock,
                      &byCurrentBits, lpGIFCVar);

    // 输出LZW_EOI
    GIF_LZW_WriteCode(file, wLZW_EOI, (LPSTR)bySubBlock,
                      &byCurrentBits, lpGIFCVar);

    if (lpGIFCVar->byLeftBits)
    {
        // 加入该字符
        bySubBlock[lpGIFCVar->wBlockNdx++] = (BYTE)lpGIFCVar->dwTempCode;

        // 判断是否超出MAX_SUBBLOCK_SIZE
        if (lpGIFCVar->wBlockNdx > MAX_SUBBLOCK_SIZE)
        {
            // 判断wByteCnt + 256是否超过MAX_BUFF_SIZE
            if ((lpGIFCVar->wByteCnt + 256) >= MAX_BUFF_SIZE)
            {
                // 输出
                file.Write(lpGIFCVar->lpDataBuff,
                           lpGIFCVar->wByteCnt);
                lpGIFCVar->lpEndBuff = lpGIFCVar->lpDataBuff;
                lpGIFCVar->wByteCnt  = 0;
            }
            bySubBlock[0]          = (BYTE)(lpGIFCVar->wBlockNdx - 1);
            memcpy(lpGIFCVar->lpEndBuff, (LPSTR)bySubBlock, lpGIFCVar->wBlockNdx);
            lpGIFCVar->lpEndBuff   += lpGIFCVar->wBlockNdx;
            lpGIFCVar->wByteCnt    += lpGIFCVar->wBlockNdx;
            lpGIFCVar->wBlockNdx    = 1;
        }
        lpGIFCVar->dwTempCode = 0UL;
        lpGIFCVar->byLeftBits = 0x00;
    }

    if (lpGIFCVar->wBlockNdx > 1)
    {
        // 判断wByteCnt + 256是否超过MAX_BUFF_SIZE
        if ((lpGIFCVar->wByteCnt + 256) >= MAX_BUFF_SIZE)
        {
            // 输出
            file.Write(lpGIFCVar->lpDataBuff,
                       lpGIFCVar->wByteCnt);
            lpGIFCVar->lpEndBuff = lpGIFCVar->lpDataBuff;
            lpGIFCVar->wByteCnt  = 0;
        }
        bySubBlock[0]          = (BYTE)(lpGIFCVar->wBlockNdx - 1);
        memcpy(lpGIFCVar->lpEndBuff, (LPSTR)bySubBlock, lpGIFCVar->wBlockNdx);
```

```
        lpGIFCVar->lpEndBuff    += lpGIFCVar->wBlockNdx;
        lpGIFCVar->wByteCnt     += lpGIFCVar->wBlockNdx;
        lpGIFCVar->wBlockNdx     = 1;
    }

    // 解除锁定
    GlobalUnlock(hTableNdx);
    GlobalUnlock(hPrefix);
    GlobalUnlock(hSuffix);

    // 释放内存
    GlobalFree(hTableNdx);
    GlobalFree(hPrefix);
    GlobalFree(hSuffix);

    // 退出
    return;
}


/*************************************************************************
 *
 * 函数名称:
 *   GIF_LZW_WriteCode()
 *
 * 参数:
 *   CFile& file             - 要保存的文件
 *   WORD wCode              - 要添加的编码
 *   LPSTR lpSubBlock        - 子块
 *   LPBYTE lpbyCurrentBits - 当前位数
 *   LPGIFC_VAR lpGIFCVar    - 指向GIFC_VAR结构的指针
 *
 * 返回值:
 *   无
 *
 * 说明:
 *   该函数用来输出一个编码。
 *
 *************************************************************************/
void WINAPI GIF_LZW_WriteCode(CFile& file, WORD wCode, LPSTR lpSubBlock,
                              LPBYTE lpbyCurrentBits,LPGIFC_VAR lpGIFCVar)
{
    // 输出该编码
    lpGIFCVar->dwTempCode |= ((DWORD)wCode << lpGIFCVar->byLeftBits);
    lpGIFCVar->byLeftBits += (*lpbyCurrentBits);

    while(lpGIFCVar->byLeftBits >= 0x08)
    {
        lpSubBlock[lpGIFCVar->wBlockNdx++] = (BYTE)lpGIFCVar->dwTempCode;

        // 判断是否超出MAX_SUBBLOCK_SIZE
```

```
        if (lpGIFCVar->wBlockNdx > MAX_SUBBLOCK_SIZE)
        {
            // 判断wByteCnt + 256是否超过MAX_BUFF_SIZE
            if ((lpGIFCVar->wByteCnt + 256) >= MAX_BUFF_SIZE)
            {
                // 输出
                file.Write(lpGIFCVar->lpDataBuff,
                           lpGIFCVar->wByteCnt);
                lpGIFCVar->lpEndBuff = lpGIFCVar->lpDataBuff;
                lpGIFCVar->wByteCnt  = 0;
            }
            lpSubBlock[0]           = (BYTE)(lpGIFCVar->wBlockNdx - 1);
            memcpy(lpGIFCVar->lpEndBuff, lpSubBlock, lpGIFCVar->wBlockNdx);
            lpGIFCVar->lpEndBuff   += lpGIFCVar->wBlockNdx;
            lpGIFCVar->wByteCnt    += lpGIFCVar->wBlockNdx;
            lpGIFCVar->wBlockNdx    = 1;
        }
        lpGIFCVar->dwTempCode >>= 8;
        lpGIFCVar->byLeftBits  -= 0x08;
    }

    // 返回
    return;
}


/*************************************************************************
 *
 * 函数名称:
 *   ReadGIF()
 *
 * 参数:
 *   CFile& file        - 要读取的文件
 *
 * 返回值:
 *   HDIB               - 成功返回DIB的句柄，否则返回NULL。
 *
 * 说明:
 *   该函数将读取指定的GIF文件。将读取的结果保存在一个未压缩
 * 编码的DIB对象中。
 *
 *************************************************************************/
HDIB WINAPI ReadGIF(CFile& file)
{

    // DIB句柄
    HDIB            hDIB;

    // DIB指针
    LPSTR           pDIB;
```

```
// 指向DIB像素的指针
LPSTR               lpDIBBits;

// 指向BITMAPINFOHEADER的指针
LPBITMAPINFOHEADER lpBIH;

// 指向BITMAPINFO的指针
LPBITMAPINFO        lpBI;

// GIF文件头
GIFHEADER           GIFH;

// GIF逻辑屏幕描述块
GIFSCRDESC          GIFS;

// GIF图像描述块
GIFIMAGE            GIFI;

// GIF图像控制扩充块
GIFCONTROL          GIFC;

// GIF图像说明扩充块
GIFPLAINTEXT        GIFP;

// GIF应用程序扩充块
GIFAPPLICATION      GIFA;

// GIF编码参数
GIFD_VAR            GIFDVar;

// 颜色数目
WORD                wColors;

// 每行字节数
WORD                wWidthBytes;

// 调色板
BYTE                byGIF_Pal[768];

// 16色系统调色板
BYTE                bySysPal16[48] = { 0,    0,    0,    0,    0, 128,
                                         0,  128,    0,    0,  128, 128,
                                       128,    0,    0,  128,    0, 128,
                                       128,  128,    0,  128,  128, 128,
                                       192,  192,  192,    0,    0, 255,
                                         0,  255,    0,    0,  255, 255,
                                       255,    0,    0,  255,    0, 255,
                                       255,  255,    0,  255,  255, 255
                                     };

// DIB大小(字节数)
```

```
DWORD              dwDIB_Size;

// 调色板大小(字节数)
WORD               wPalSize;

// 字节变量
BYTE               byTemp;

// 内存句柄
HANDLE             hSrcBuff;
HANDLE             hTemp;

// 内存指针
LPSTR              lpTemp;

// 字变量
WORD               wTemp;

// 循环变量
WORD               wi;

// 标签
BYTE               byLabel;

// 块大小
BYTE               byBlockSize;

// 读取GIF文件头
file.Read((LPSTR)&GIFH, sizeof(GIFH));

// 判断是否是GIF文件
if (memicmp((LPSTR)GIFH.bySignature,"GIF",3) != 0)
{
    // 非GIF文件，返回NULL
    return NULL;
}

// 判断版本号是否正确
if ((memicmp((LPSTR)GIFH.byVersion,"87a",3) != 0) &&
    (memicmp((LPSTR)GIFH.byVersion,"89a",3) != 0))
{
    // 不支持该版本，返回NULL
    return NULL;
}

// 读取GIF逻辑屏幕描述块
file.Read((LPSTR)&GIFS, 7);

// 获取调色板的位数
GIFDVar.wBits = (WORD)GIFS.GlobalFlag.PalBits + 1;
```

```
// 判断是否有全局调色板
if (GIFS.GlobalFlag.GlobalPal)
{
    // 赋初值
    memset((LPSTR)byGIF_Pal,0,768);

    // 全局调色板大小
    wPalSize = 3 * (1 << GIFDVar.wBits);

    // 读取全局调色板
    file.Read((LPSTR)byGIF_Pal,wPalSize);
}

// 读取下一个字节
file.Read((LPSTR)&byTemp,1);

// 对每一个描述块循环
while(TRUE)
{
    // 判断是否是图像描述块
    if (byTemp == 0x2C)
    {
        // 是图像描述块，退出循环
        break;
    }

    // 判断是否是GIF扩展块
    if (byTemp==0x21)
    {
        // 是GIF扩展块

        // 分配内存
        hTemp  = GlobalAlloc(GHND,(DWORD)MAX_BUFF_SIZE);

        // 锁定内存
        lpTemp = (LPSTR) GlobalLock(hTemp);

        // 读取下一个字节
        file.Read((LPSTR)&byLabel, 1);

        // 针对各种扩充块，进行分别处理
        switch(byLabel)
        {
            case 0xF9:
            {
                // 图像控制扩充块
                file.Read((LPSTR)&GIFC, 6);

                // 跳出
                break;
            }
```

```
case 0x01:
{
    // 图像说明扩充块
    file.Read((LPSTR)&GIFP,sizeof(GIFP));

    // 读取扩充块大小
    file.Read((LPSTR)&byBlockSize,1);

    // 当byBlockSize > 0时循环读取
    while(byBlockSize)
    {
        // 读取图像说明扩充块（这里没有进行任何处理）
        file.Read(lpTemp,byBlockSize);

        // 读取扩充块大小
        file.Read((LPSTR)&byBlockSize,1);
    }

    // 跳出
    break;
}
case 0xFE:
{
    // 注释说明扩充块

    // 读取扩充块大小
    file.Read((LPSTR)&byBlockSize,1);

    // 当byBlockSize > 0时循环读取
    while(byBlockSize)
    {
        // 读取注释说明扩充块（这里没有进行任何处理）
        file.Read(lpTemp,byBlockSize);

        // 读取扩充块大小
        file.Read((LPSTR)&byBlockSize,1);
    }

    // 跳出
    break;
}
case 0xFF:
{
    // 应用程序扩充块
    file.Read((LPSTR)&GIFA, sizeof(GIFA));

    // 读取扩充块大小
    file.Read((LPSTR)&byBlockSize,1);

    // 当byBlockSize > 0时循环读取
    while(byBlockSize)
```

```
                        {
                            // 读取应用程序扩充块（这里没有进行任何处理）
                            file.Read(lpTemp, byBlockSize);

                            // 读取扩充块大小
                            file.Read((LPSTR)&byBlockSize, 1);
                        }

                        // 跳出
                        break;
                    }
                    default:
                    {
                        // 忽略未知的扩充块

                        // 读取扩充块大小
                        file.Read((LPSTR)&byBlockSize, 1);

                        // 当byBlockSize > 0时循环读取
                        while(byBlockSize)
                        {
                            // 读取未知的扩充块（这里没有进行任何处理）
                            file.Read(lpTemp, byBlockSize);

                            // 读取扩充块大小
                            file.Read((LPSTR)&byBlockSize, 1);
                        }

                        // 跳出
                        break;
                    }

                    // 释放内存
                    GlobalUnlock(hTemp);
                    GlobalFree(hTemp);
                }
            }

            // 读取下一个字节
            file.Read((LPSTR)&byTemp, 1);
        }

        // 读取GIF图像描述块
        file.Read((LPSTR)&GIFI, 9);

        // 获取图像宽度
        GIFDVar.wWidth          = GIFI.wWidth;

        // 获取图像高度
        GIFDVar.wDepth          = GIFI.wDepth;
```

```
// 判断是否有区域调色板
if (GIFI.LocalFlag.LocalPal)
{
    // 赋初值
    memset((LPSTR)byGIF_Pal, 0, 768);

    // 读取区域调色板位数
    GIFDVar.wBits = (WORD)GIFI.LocalFlag.PalBits + 1;

    // 区域调色板大小
    wPalSize      = 3 * (1 << GIFDVar.wBits);

    // 读取区域调色板
    file.Read((LPSTR)byGIF_Pal,wPalSize);
}

// 给GIFDVar成员赋值
GIFDVar.wBits          = ((GIFDVar.wBits==1) ? 1 :
                          (GIFDVar.wBits<=4) ? 4 : 8);

GIFDVar.wLineBytes     = (WORD)BYTE_WBYTES((DWORD)GIFDVar.wWidth *
                                  (DWORD)GIFDVar.wBits);
GIFDVar.bEOF           = FALSE;

// 交错方式
GIFDVar.bInterlace     = (GIFI.LocalFlag.Interlace ? TRUE : FALSE);

// 每行字节数
wWidthBytes            = (WORD)DWORD_WBYTES((DWORD)GIFDVar.wWidth *
                                  (DWORD)GIFDVar.wBits);

// 颜色数目
wColors                = 1 << GIFDVar.wBits;

// 调色板大小
wPalSize               = wColors * sizeof(RGBQUAD);

// 计算DIB长度（字节）
dwDIB_Size             = sizeof(BITMAPINFOHEADER) + wPalSize
                         + GIFDVar.wDepth * wWidthBytes;

// 为DIB分配内存
hDIB = (HDIB) ::GlobalAlloc(GMEM_MOVEABLE | GMEM_ZEROINIT, dwDIB_Size);

if (hDIB == 0)
{
    // 内存分配失败，返回NULL。
    return NULL;
}

// 锁定
```

```
pDIB = (LPSTR) ::GlobalLock((HGLOBAL) hDIB);

////////////////////////////////////////////////////////////////////////////
// 设置BITMAPINFOHEADER

// 赋值
lpBIH = (LPBITMAPINFOHEADER) pDIB;
lpBI  = (LPBITMAPINFO) pDIB;

// 给lpBIH成员赋值
lpBIH->biSize = (DWORD)sizeof(BITMAPINFOHEADER);
lpBIH->biWidth = (DWORD)GIFDVar.wWidth;
lpBIH->biHeight = (DWORD)GIFDVar.wDepth;
lpBIH->biPlanes = 1;
lpBIH->biBitCount = GIFDVar.wBits;
lpBIH->biCompression = BI_RGB;
lpBIH->biSizeImage = (DWORD)wWidthBytes * GIFDVar.wDepth;
lpBIH->biXPelsPerMeter = 0;
lpBIH->biYPelsPerMeter = 0;
lpBIH->biClrUsed = wColors;
lpBIH->biClrImportant = 0;

////////////////////////////////////////////////////////////////////////////
// 设置调色板

// 判断是否指定全局或区域调色板
if (GIFS.GlobalFlag.GlobalPal || GIFI.LocalFlag.LocalPal)
{
    wTemp = 0;
    for(wi=0; wi<wColors; wi++)
    {
        lpBI->bmiColors[wi].rgbRed      = byGIF_Pal[wTemp++];
        lpBI->bmiColors[wi].rgbGreen    = byGIF_Pal[wTemp++];
        lpBI->bmiColors[wi].rgbBlue     = byGIF_Pal[wTemp++];
        lpBI->bmiColors[wi].rgbReserved = 0x00;
    }
}
else
{
    // 没有指定全局和区域调色板，采用系统调色板

    // 按照颜色数目来分别给调色板赋值
    switch(wColors)
    {
        case  2:
        {
            // 单色位图

            lpBI->bmiColors[0].rgbRed         = 0x00;
            lpBI->bmiColors[0].rgbGreen       = 0x00;
            lpBI->bmiColors[0].rgbBlue        = 0x00;
```

```
            lpBI->bmiColors[0].rgbReserved      = 0x00;
            lpBI->bmiColors[1].rgbRed           = 0xFF;
            lpBI->bmiColors[1].rgbGreen         = 0xFF;
            lpBI->bmiColors[1].rgbBlue          = 0xFF;
            lpBI->bmiColors[1].rgbReserved      = 0x00;

            // 跳出
            break;
        }
        case  16:
        {
            // 16色位图

            wTemp = 0;
            for(wi=0;wi<wColors;wi++)
            {
                lpBI->bmiColors[wi].rgbRed      = bySysPal16[wTemp++];
                lpBI->bmiColors[wi].rgbGreen    = bySysPal16[wTemp++];
                lpBI->bmiColors[wi].rgbBlue     = bySysPal16[wTemp++];
                lpBI->bmiColors[wi].rgbReserved = 0x00;
            }

            // 跳出
            break;
        }
        case 256:
        {
            // 256色位图
            for(wi=0; wi<wColors; wi++)
            {
                lpBI->bmiColors[wi].rgbRed      = (BYTE)wi;
                lpBI->bmiColors[wi].rgbGreen    = (BYTE)wi;
                lpBI->bmiColors[wi].rgbBlue     = (BYTE)wi;
                lpBI->bmiColors[wi].rgbReserved = 0x00;
            }

            // 跳出
            break;
        }
    }
}

//////////////////////////////////////////////////////////////////////////////
// 解码

// 获取编码数据长度
GIFDVar.dwDataLen  = (DWORD) (file.GetLength() - file.GetPosition());

// 计算内存大小（最大不超过MAX_BUFF_SIZE）
GIFDVar.wMemLen     = ((GIFDVar.dwDataLen > (DWORD)MAX_BUFF_SIZE) ?
                        (WORD)MAX_BUFF_SIZE : (WORD)GIFDVar.dwDataLen);
```

```
    // 分配内存
    hSrcBuff = GlobalAlloc(GHND, (DWORD)GIFDVar.wMemLen);

    // 锁定内存
    GIFDVar.lpDataBuff = (LPSTR)GlobalLock(hSrcBuff);

    // 读取编码数据
    ReadSrcData(file,&GIFDVar.wMemLen,&GIFDVar.dwDataLen,
                GIFDVar.lpDataBuff,&GIFDVar.bEOF);

    // 缓冲区起始位置
    GIFDVar.lpBgnBuff  = GIFDVar.lpDataBuff;

    // 缓冲区中止位置
    GIFDVar.lpEndBuff  = GIFDVar.lpBgnBuff + GIFDVar.wMemLen;

    // 计算DIB中像素位置
    lpDIBBits = (LPSTR) FindDIBBits(pDIB);

    // 解码
    DecodeGIF_LZW(file, lpDIBBits, &GIFDVar, wWidthBytes);

    // 释放内存
    GlobalUnlock(hSrcBuff);
    GlobalFree(hSrcBuff);

    // 返回DIB句柄
    return hDIB;
}


/*************************************************************************
 *
 * 函数名称:
 *   ReadSrcData()
 *
 * 参数:
 *   CFile& file              - 源GIF文件
 *   LPWORD lpwMemLen         - 缓冲区长度（指针）
 *   LPDWORD lpdwDataLen      - 剩余数据长度（指针）
 *   LPSTR lpSrcBuff          - 缓冲区指针
 *   LPBOOL lpbEOF            - 结束标志
 *
 * 返回值:
 *   无
 *
 * 说明:
 *   该函数用来读取指定GIF文件中的图像编码，每次最多读取MAX_BUFF_SIZE
 * 字节，是否读完由标志lpbEOF指定。
 *
```

```
 ***************************************************************************/
void WINAPI ReadSrcData(CFile& file, LPWORD lpwMemLen, LPDWORD lpdwDataLen,
                        LPSTR lpSrcBuff, LPBOOL lpbEOF)
{
    // 判断数据长度是否仍然大于内存大小
    if ((*lpdwDataLen) > (DWORD)(*lpwMemLen))
    {
        // 数据长度大于内存大小，表示没有解码完

        // 数据长度减内存大小
        (*lpdwDataLen) -= (DWORD)(*lpwMemLen);
    }
    else
    {
        // 数据长度不大于内存大小，表示解码将要完成

        // 内存大小就是剩余数据长度
        (*lpwMemLen)    = (WORD)(*lpdwDataLen);

        // EOF标志设置为TRUE
        (*lpbEOF)       = TRUE;
    }

    // 读取编码数据
    file.Read(lpSrcBuff, (*lpwMemLen));

    // 返回
    return;
}

/***************************************************************************
 *
 * 函数名称:
 *   DecodeGIF_LZW()
 *
 * 参数:
 *   CFile& file              - 源GIF文件
 *   LPSTR lpDIBBits          - 指向要保存的DIB图像指针
 *   LPGIFD_VAR lpGIFDVar     - 指向GIFC_VAR结构的指针
 *   WORD wWidthBytes         - 每行图像字节数
 *
 * 返回值:
 *   无
 *
 * 说明:
 *   该函数对指定GIF_LZW编码数据进行解码。
 *
 ***************************************************************************/
void WINAPI DecodeGIF_LZW(CFile& file, LPSTR lpDIBBits,
                          LPGIFD_VAR lpGIFDVar,WORD wWidthBytes)
{
```

```
// 指向编码后图像数据的指针
BYTE *  lpDst;

// 内存分配句柄
HANDLE hPrefix;
HANDLE hSuffix;
HANDLE hStack;
HANDLE hImage;

// 用于字串表搜索的索引
LPWORD lpwPrefix;
LPBYTE lpbySuffix;
LPBYTE lpbyStack;
LPBYTE lpbyStackBgn;

// 指向图像当前行解码结果的指针
LPSTR  lpImageBgn;

// 指向当前编码像素的指针
LPSTR  lpImage;

// 计算当前数据图像的偏移量
DWORD  dwDataNdx;

// LZW_CLEAR
WORD   wLZW_CLEAR;

// LZW_EOI
WORD   wLZW_EOI;

// LZW_MinCodeLen
BYTE   byLZW_MinCodeLen;

// 字串表索引
WORD   wNowTableNdx;
WORD   wTopTableNdx;

// 当前图像的行数
WORD   wRowNum;

// 计数
WORD   wWidthCnt;
WORD   wBitCnt;
WORD   wRowCnt;

// 循环变量
WORD   wi;

// 交错方式存储时每次增加的行数
WORD   wIncTable[5]  = { 8,8,4,2,0 };
```

```
// 交错方式存储时起始行数
WORD    wBgnTable[5]  = { 0,4,2,1,0 };

// 块大小
BYTE    byBlockSize;

// 块索引
BYTE    byBlockNdx;

DWORD   dwData;

// 当前编码
WORD    wCode;

// 上一个编码
WORD    wOldCode;

// 临时索引
WORD    wTempNdx;

WORD    wCodeMask[13]  = {0x0000,
                          0x0001,0x0003,0x0007,0x000F,
                          0x001F,0x003F,0x007F,0x00FF,
                          0x01FF,0x03FF,0x07FF,0x0FFF
                          };

BYTE    byLeftBits;
BYTE    byFirstChar;
BYTE    byCode;
BYTE    byCurrentBits;
BYTE    byPass;

// 临时字节变量
BYTE    byTempChar;

// 给字串表分配内存
hPrefix         = GlobalAlloc(GHND, (DWORD)(MAX_TABLE_SIZE<<1));
hSuffix         = GlobalAlloc(GHND, (DWORD)MAX_TABLE_SIZE);
hStack          = GlobalAlloc(GHND, (DWORD)MAX_TABLE_SIZE);
hImage          = GlobalAlloc(GHND, (DWORD)wWidthBytes);

// 锁定内存
lpwPrefix       = (LPWORD)GlobalLock(hPrefix);
lpbySuffix      = (LPBYTE)GlobalLock(hSuffix);
lpbyStack       = (LPBYTE)GlobalLock(hStack);
lpbyStackBgn    = lpbyStack;
lpImage         = (LPSTR)GlobalLock(hImage);
lpImageBgn      = lpImage;

// 读取GIF LZW最小编码大小
byLZW_MinCodeLen = *lpGIFDVar->lpBgnBuff++;
```

```
byCurrentBits    = byLZW_MinCodeLen + (BYTE)0x01;

// 计算LZW_CLEAR
wLZW_CLEAR       = 1 << byLZW_MinCodeLen;

// 计算LZW_EOI
wLZW_EOI         = wLZW_CLEAR + 1;

// 计算字串表索引
wNowTableNdx     = wLZW_CLEAR + 2;
wTopTableNdx     = 1 << byCurrentBits;

// 赋初值
dwData           = 0UL;
wBitCnt          = lpGIFDVar->wBits;
wRowNum          = 0;
wRowCnt          = 1;
wWidthCnt        = 0;
wCode            = 0;
wOldCode         = 0xFFFF;
byBlockSize      = 0x01;
byBlockNdx       = 0x00;
byLeftBits       = 0x00;
byTempChar       = 0x00;
byPass           = 0x00;

// 读取下一个编码
while(byLeftBits < byCurrentBits)
{
    // 读取下一个字符

    // 判断是否读完一个数据块
    if (++byBlockNdx == byBlockSize)
    {
        // 读取下一个数据块
        byBlockSize = *lpGIFDVar->lpBgnBuff++;
        byBlockNdx  = 0x00;

        // 判断是否读完
        if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
            !lpGIFDVar->bEOF)
        {
            // 读取下一个数据块
            ReadSrcData(file,&lpGIFDVar->wMemLen,
                        &lpGIFDVar->dwDataLen,
                        lpGIFDVar->lpDataBuff,&lpGIFDVar->bEOF);

            // 指针重新赋值
            lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
            lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff + lpGIFDVar->wMemLen;
```

```
            }
        }

        // 下一个字符
        byCode      = *lpGIFDVar->lpBgnBuff++;

        // 移位
        dwData      |= ((DWORD)byCode << byLeftBits);
        byLeftBits += 0x08;

        // 判断是否读完
        if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
            !lpGIFDVar->bEOF)
        {
            // 读取下一个数据块
            ReadSrcData(file,&lpGIFDVar->wMemLen,
                        &lpGIFDVar->dwDataLen,
                        lpGIFDVar->lpDataBuff,&lpGIFDVar->bEOF);

            // 指针重新赋值
            lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
            lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff + lpGIFDVar->wMemLen;
        }
    }

    wCode       = (WORD)dwData & wCodeMask[byCurrentBits];
    dwData    >>= byCurrentBits;
    byLeftBits -= byCurrentBits;

    // 解码
    while(wCode != wLZW_EOI)
    {
        // 当前编码不是LZW_EOI码

        // 判断是否是LZW_CLEAR码
        if (wCode == wLZW_CLEAR)
        {
            // 是LZW_CLEAR，清除字串表

            // 重新初始化字串表
            for(wi = 0; wi < wLZW_CLEAR; wi++)
            {
                *(lpwPrefix  + wi)  = 0xFFFF;
                *(lpbySuffix + wi)  = (BYTE)wi;
            }

            for(wi = wNowTableNdx; wi < MAX_TABLE_SIZE; wi++)
            {
                *(lpwPrefix+wi)  = 0xFFFF;
                *(lpbySuffix+wi) = 0x00;
            }
```

```
byCurrentBits = byLZW_MinCodeLen + (BYTE)0x01;
wNowTableNdx  = wLZW_CLEAR + 2;
wTopTableNdx  = 1 << byCurrentBits;
wOldCode      = 0xFFFF;

// 获取下一个编码
while(byLeftBits < byCurrentBits)
{
    // 读取下一个字符

    // 判断是否读完一个数据块
    if (++byBlockNdx == byBlockSize)
    {
        // 读取下一个数据块
        byBlockSize = *lpGIFDVar->lpBgnBuff++;
        byBlockNdx  = 0x00;

        // 判断是否读完
        if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
            !lpGIFDVar->bEOF)
        {
            // 读取下一个数据块
            ReadSrcData(file, &lpGIFDVar->wMemLen,
                        &lpGIFDVar->dwDataLen,
                        lpGIFDVar->lpDataBuff,
                        &lpGIFDVar->bEOF);

            // 指针重新赋值
            lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
            lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff +
                                   lpGIFDVar->wMemLen;
        }
    }
    byCode      = *lpGIFDVar->lpBgnBuff++;
    dwData     |= ((DWORD)byCode << byLeftBits);
    byLeftBits += 0x08;

    // 判断是否读完
    if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
        !lpGIFDVar->bEOF)
    {
        // 读取下一个数据块
        ReadSrcData(file,&lpGIFDVar->wMemLen,
                    &lpGIFDVar->dwDataLen,
                    lpGIFDVar->lpDataBuff,&lpGIFDVar->bEOF);

        // 指针重新赋值
        lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
        lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff + lpGIFDVar->wMemLen;
    }
```

```
}
wCode        = (WORD)dwData & wCodeMask[byCurrentBits];
dwData     >>= byCurrentBits;
byLeftBits -= byCurrentBits;

// 判断编码是否为LZW_EOI
if (wCode!=wLZW_EOI)
{
    // 这里没有用到lpbyStack[0]
    lpbyStack  ++;

    // 将数据压入堆栈
    while((*(lpwPrefix+wCode)) != 0xFFFF)
    {
        *lpbyStack++ = *(lpbySuffix+wCode);
        wCode        = *(lpwPrefix+wCode);
    }
    *lpbyStack   = *(lpbySuffix+wCode);
    byFirstChar = *lpbyStack;

    // 输出数据
    while(lpbyStack>lpbyStackBgn)
    {
        byTempChar |= (*lpbyStack-- << (8-wBitCnt));

        if (wBitCnt==8)
        {
            *lpImage++ = byTempChar;
            byTempChar = 0x00;
            wBitCnt    = lpGIFDVar->wBits;
        }
        else
        {
            wBitCnt   += lpGIFDVar->wBits;
        }

        wWidthCnt  ++;

        if (wWidthCnt==lpGIFDVar->wWidth)
        {
            if (wBitCnt!=lpGIFDVar->wBits)
            {
                *lpImage   = byTempChar;
                byTempChar = 0x00;
                wBitCnt    = lpGIFDVar->wBits;
            }

            // 图像当前行偏移量
            dwDataNdx = (DWORD)(lpGIFDVar->wDepth - 1 - wRowNum) *
                    (DWORD)wWidthBytes;
```

```
                    // 图像当前行起始位置
                    lpDst = (BYTE *)lpDIBBits + dwDataNdx;

                    // 赋值
                    memcpy(lpDst, lpImageBgn, wWidthBytes);

                    lpImage   = lpImageBgn;

                    // 判断是否按照交错方式保存
                    if (lpGIFDVar->bInterlace)
                    {
                        // 交错方式

                        // 计算下一行的行号
                        wRowNum += wIncTable[byPass];
                        if (wRowNum >= lpGIFDVar->wDepth)
                        {
                            byPass ++;
                            wRowNum = wBgnTable[byPass];
                        }
                    }
                    else
                    {
                        // 非交错方式，行号直接加1
                        wRowNum ++;
                    }
                    wWidthCnt = 0;
                }
            }
        }
    }
    else
    {
        // 这里没有用到lpbyStack[0]
        lpbyStack  ++;

        // 判断字符串是否在字串表中
        if (wCode < wNowTableNdx)
        {
            // 不在字串表中
            wTempNdx    = wCode;
        }
        else
        {
            // 在字串表中
            wTempNdx      = wOldCode;
            *lpbyStack++ = byFirstChar;
        }

        // 将数据压入堆栈
        while((*(lpwPrefix+wTempNdx)) != 0xFFFF)
```

```
{
    *lpbyStack++ = *(lpbySuffix+wTempNdx);
    wTempNdx     = *(lpwPrefix+wTempNdx);
}
*lpbyStack   = *(lpbySuffix+wTempNdx);
byFirstChar = *lpbyStack;

// 将字符串添加到字串表中
*(lpwPrefix+wNowTableNdx)  = wOldCode;
*(lpbySuffix+wNowTableNdx) = byFirstChar;
if (++wNowTableNdx==wTopTableNdx && byCurrentBits<12)
{
    byCurrentBits ++;
    wTopTableNdx   = 1 << byCurrentBits;
}

// 输出数据
while(lpbyStack>lpbyStackBgn)
{
    byTempChar |= (*lpbyStack-- << (8-wBitCnt));
    if (wBitCnt==8)
    {
        *lpImage++ = byTempChar;
        byTempChar = 0x00;
        wBitCnt    = lpGIFDVar->wBits;
    }
    else
    {
        wBitCnt   += lpGIFDVar->wBits;
    }

    wWidthCnt  ++;
    if (wWidthCnt==lpGIFDVar->wWidth)
    {
        if (wBitCnt!=lpGIFDVar->wBits)
        {
            *lpImage    = byTempChar;
            byTempChar = 0x00;
            wBitCnt    = lpGIFDVar->wBits;
        }

        // 图像当前行偏移量
        dwDataNdx = (DWORD)(lpGIFDVar->wDepth - 1 - wRowNum) *
                (DWORD)wWidthBytes;

        // 图像当前行起始位置
        lpDst = (BYTE *)lpDIBBits + dwDataNdx;

        // 赋值
        memcpy(lpDst, lpImageBgn, wWidthBytes);
```

```
                lpImage   = lpImageBgn;

                // 判断是否按照交错方式保存
                if (lpGIFDVar->bInterlace)
                {
                    // 交错方式

                    // 计算下一行的行号
                    wRowNum += wIncTable[byPass];
                    if (wRowNum >= lpGIFDVar->wDepth)
                    {
                        byPass ++;
                        wRowNum = wBgnTable[byPass];
                    }
                }
                else
                {
                    // 非交错方式，行号直接加1
                    wRowNum ++;
                }
                wWidthCnt  = 0;
            }
        }
    }
    wOldCode = wCode;

    // 读取下一个编码
    while(byLeftBits < byCurrentBits)
    {
        // 读取下一个字符

        // 判断是否读完一个数据块
        if (++byBlockNdx == byBlockSize)
        {
            // 读取下一个数据块
            byBlockSize = *lpGIFDVar->lpBgnBuff++;
            byBlockNdx  = 0x00;

            // 判断是否读完
            if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
                !lpGIFDVar->bEOF)
            {
                // 读取下一个数据块
                ReadSrcData(file,&lpGIFDVar->wMemLen,
                            &lpGIFDVar->dwDataLen,
                            lpGIFDVar->lpDataBuff,&lpGIFDVar->bEOF);

                // 指针重新赋值
                lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
                lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff + lpGIFDVar->wMemLen;
            }
```

```
            }
            byCode      = *lpGIFDVar->lpBgnBuff++;
            dwData     |= ((DWORD)byCode << byLeftBits);
            byLeftBits += 0x08;

            // 判断是否读完
            if ((lpGIFDVar->lpBgnBuff == lpGIFDVar->lpEndBuff) &&
                !lpGIFDVar->bEOF)
            {
                // 读取下一个数据块
                ReadSrcData(file,&lpGIFDVar->wMemLen,
                            &lpGIFDVar->dwDataLen,
                            lpGIFDVar->lpDataBuff,&lpGIFDVar->bEOF);

                // 指针重新赋值
                lpGIFDVar->lpBgnBuff = lpGIFDVar->lpDataBuff;
                lpGIFDVar->lpEndBuff = lpGIFDVar->lpBgnBuff + lpGIFDVar->wMemLen;
            }
        }
        wCode       = (WORD)dwData & wCodeMask[byCurrentBits];
        dwData    >>= byCurrentBits;
        byLeftBits -= byCurrentBits;
    }

    // 释放内存
    GlobalUnlock(hPrefix);
    GlobalUnlock(hSuffix);
    GlobalUnlock(hStack);
    GlobalFree(hPrefix);
    GlobalFree(hSuffix);
    GlobalFree(hStack);

    // 返回
    return;
}
```

利用上面的函数可以将当前打开的图像存成 GIF 格式。下面我们来编写菜单“LZW 编码”子菜单中的“保存为 GIF 文件”菜单项（如图 11－9 所示）的单击事件。

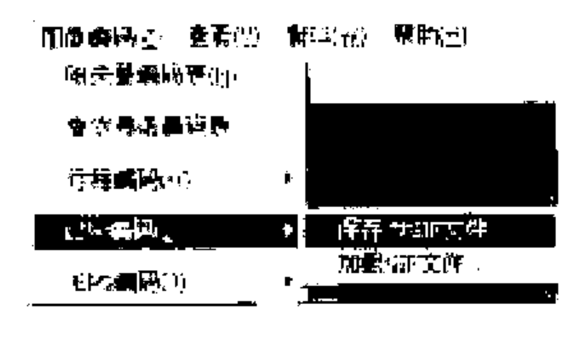


图 11－9　LZW 编码菜单

```
void CCh1_1View::OnCodeLzw()
{
    // 对当前图像进行GIF-LZW编码（存为GIF格式文件）

    // 获取文档
    CCh1_1Doc* pDoc = GetDocument();

    // 指向DIB的指针
    LPSTR    lpDIB;

    // 指向DIB像素指针
    LPSTR     lpDIBBits;

    // 锁定DIB
    lpDIB = (LPSTR) ::GlobalLock((HGLOBAL) pDoc->GetHDIB());

    // 找到DIB图像像素起始位置
    lpDIBBits = ::FindDIBBits(lpDIB);

    // 判断是否超过256色
    if (::DIBNumColors(lpDIB) > 256)
    {
        // 提示用户
        MessageBox("目前只支持< 256色位图的GIF-LZW编码！", "系统提示" ,
            MB_ICONINFORMATION | MB_OK);

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 返回
        return;
    }

    // 更改光标形状
    BeginWaitCursor();

    // 文件保存路径
    CString strFilePath;

    // 指定是否以交错方式保存GIF文件
    BOOL     bInterlace;

    // 初始化文件名为原始文件名
    strFilePath = pDoc->GetPathName();

    // 更改后缀为GIF
    if (strFilePath.Right(4).CompareNoCase(".BMP") == 0)
    {
        // 更改后缀为GIF
        strFilePath = strFilePath.Left(strFilePath.GetLength()-3) + "GIF";
    }
```

```
else
{
    // 直接添加后缀GIF
    strFilePath += ".GIF";
}

// 创建SaveAs对话框
CDlgCodeGIF dlg;

dlg.m_strFilePath = strFilePath;
dlg.m_bInterlace = FALSE;

// 提示用户选择保存的路径
if (dlg.DoModal() != IDOK)
{
    // 解除锁定
    ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

    // 恢复光标
    EndWaitCursor();

    // 返回
    return;
}

// 获取用户指定的文件路径
strFilePath = dlg.m_strFilePath;
bInterlace = dlg.m_bInterlace;

// CFile和CFileException对象
CFile file;
CFileException fe;

// 尝试创建指定的GIF文件
if (!file.Open(strFilePath, CFile::modeCreate |
  CFile::modeReadWrite | CFile::shareExclusive, &fe))
{
    // 提示用户
    MessageBox("打开指定GIF文件时失败！", "系统提示",
        MB_ICONINFORMATION | MB_OK);

    // 返回
    return;
}

// 调用DIBToGIF()函数将当前的DIB保存为GIF文件
if (::DIBToGIF(lpDIB, file, bInterlace))
{
    // 提示用户
    MessageBox("成功保存为GIF文件！", "系统提示",
        MB_ICONINFORMATION | MB_OK);
```

```
        }
        else
        {
            // 提示用户
            MessageBox("保存为GIF文件失败！", "系统提示" ,
                MB_ICONINFORMATION | MB_OK);
        }

        // 解除锁定
        ::GlobalUnlock((HGLOBAL) pDoc->GetHDIB());

        // 恢复光标
        EndWaitCursor();
    }
```

菜单“LZW 编码”子菜单中的“加载为 GIF 文件”菜单项（如图 11－9 所示）的单击事件如下。

```
    void CCh1_1View::OnCodeIlzw()
    {
        // 加载GIF文件

        // 文件路径
        CString strFilePath;

        // 创建Open对话框
        CFileDialog dlg(TRUE, "GIF", NULL, OFN_HIDEREADONLY | OFN_OVERWRITEPROMPT,
            "GIF图像文件 (*.GIF) | *.GIF|所有文件 (*.*) | *.*||", NULL);

        // 提示用户选择保存的路径
        if (dlg.DoModal() != IDOK)
        {
            // 返回
            return;
        }

        // 获取用户指定的文件路径
        strFilePath = dlg.GetPathName();

        // CFile和CFileException对象
        CFile file;
        CFileException fe;

        // 尝试打开指定的GIF文件
        if (!file.Open(strFilePath, CFile::modeRead | CFile::shareDenyWrite, &fe))
        {
            // 提示用户
            MessageBox("打开指定GIF文件时失败！", "系统提示" ,
                MB_ICONINFORMATION | MB_OK);

            // 返回
            return;
```

```
        }

        // 调用ReadGIF()函数读取指定的GIF文件
        HDIB hDIB = ::ReadGIF(file);

        if (hDIB != NULL)
        {
            // 提示用户
            //MessageBox("成功读取GIF文件！", "系统提示",
            //  MB_ICONINFORMATION | MB_OK);

            // 获取文档
            CCh1_1Doc* pDoc = GetDocument();

            // 替换DIB，同时释放旧DIB对象
            pDoc->ReplaceHDIB(hDIB);

            // 更新DIB大小和调色板
            pDoc->InitDIBData();

            // 设置脏标记
            pDoc->SetModifiedFlag(TRUE);

            // 重新设置滚动视图大小
            SetScrollSizes(MM_TEXT, pDoc->GetDocSize());

            // 实现新的调色板
            OnDoRealize((WPARAM)m_hWnd,0);

            // 更新视图
            pDoc->UpdateAllViews(NULL);
        }
        else
        {
            // 提示用户
            MessageBox("读取GIF文件失败！", "系统提示",
                MB_ICONINFORMATION | MB_OK);
        }

    }
```

11.5 JPEG 编码

JPEG（联合图像专家组，Joint Photographic Experts Group）是由国际标准组织（ISO，International Standardization Organization）和国际电话电报咨询委员会（CCITT，Consultation Committee of the International Telephone and Telegraph）为静态图像所建立的第一个国际数字图像压缩标准。和相同图像质量的其他常用图像文件格式（如 GIF，TIFF，PCX）相比，JPEG

是目前静态图像中压缩比最高的。它采用有损压缩方式来进行图像压缩，但失真的程度非常小，肉眼几乎无法辨认。当然，JEPG 也支持无损方式的图像压缩，但其压缩比就不能达到那么高。由于其高压缩比，JPEG 目前被广泛地应用于多媒体和网络程序中，其中 HTML 语法中标准的图像文件格式之一就是 JPEG 文件格式（另一种是 GIF 文件格式）。

JPEG 编码可以分为几种模式，有基于离散余弦变换（DCT）的有失真模式，也有使用预测器的无失真模式。按照编码的次序，又可以将 JEPG 编码分成顺序式编码（Sequential Encoding）和递增式编码（Progressive Encoding）。前者是将图像从左到右、从上到下顺序编码；而后者是将图像分次处理，从模糊到清晰的方式来传输图像（这与 GIF 图像文件的交错方式类似）。基于 DCT 变换的 JEPG，可以分为仅能接受每像素点以 8 位表示的基本图像处理（Baseline Process）模式和接受每个像素点为 8 位或 12 位表示的扩展图像处理（Extended Process）模式。其中最常用的 JEPG 编码是基于 DCT 变换的顺序型基本图像处理模式，以下是针对这种格式进行讨论。

11.5.1 理论基础

JPEG 的编码的流程如图 11－10 所示。

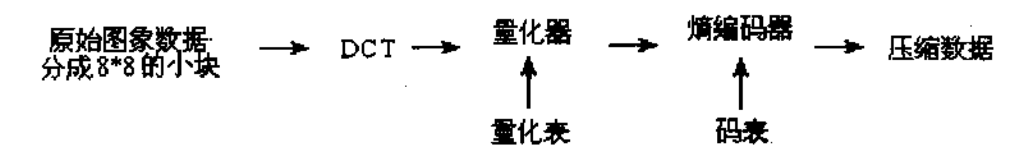


图 11－10 JEPG 编码器流程

JPEG 的解码的流程如图 11－12 所示。

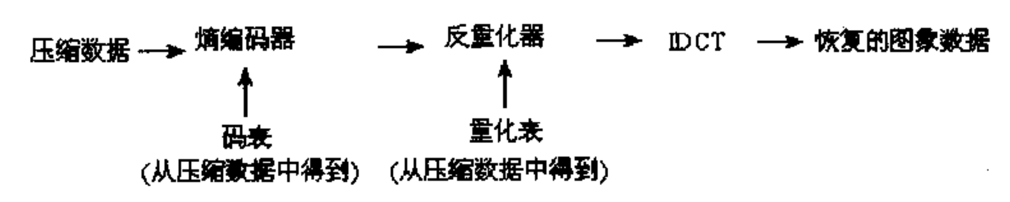


图 11－11 JEPG 解码器流程

8×8 的图像经过 DCT 变换后，其低频分量都集中在左上角，高频分量分布在右下角（DCT 变换实际上是空间域的低通滤波器）。由于该低频分量包含了图像的主要信息（如亮度），高频与之相比，就显得不那么重要了，所以在编码时，可以忽略图像的高频分量，从而达到压缩的目的。要将高频分量去掉，就要用到量化，它是产生信息损失的根源。这里的量化操作，就是将某一个值除以量化表中对应的值，由于量化表左上角的值较小，右上角的值较大，这样就起到了保持低频分量、抑制高频分量的目的。

JPEG 使用的颜色系统是我们前面介绍的 YC_bC_r 系统。其中 Y 分量代表了亮度信息，C_bC_r 分量代表了色调信息。由于 Y 分量更重要一些，所以应对 Y 采用细量化，对 C_bC_r 采用粗量化，这样可以提高压缩比。所以上面所说的量化表通常有两张：一张是针对 Y 分量的；一张是针对 C_bC_r 分量的。

由于经过DCT变换后，低频分量集中在左上角，其中F（0，0）（即第一行第一列元素）代表了直流（DC）系数，即8*8子块的平均值，要对它单独编码。由于两个相邻的8*8子块的DC系数相差很小，所以对它们采用差分编码DPCM，可以提高压缩比，也就是说对相邻的子块DC系数的差值进行编码。8*8的其他63个元素是交流（AC）系数，采用行程编码。这里出现一个问题：这63个系数应该按照怎么样的顺序排列？

为了保证低频分量先出现，高频分量后出现，以增加行程中连续“0”的个数，这63个元素采用了“之”字型（Zig-Zag）的排列方法。如图11－12所示。

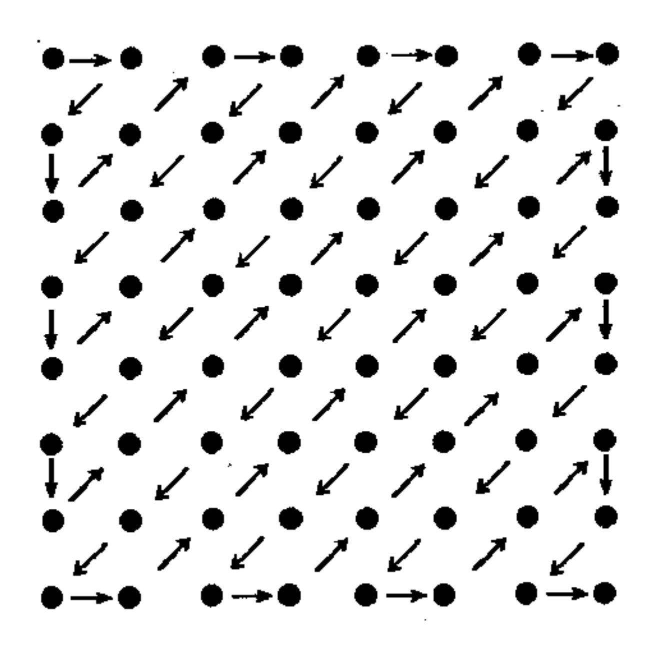


图11－12　Zig-Zag

这63个AC系数行程编码的码字用两个字节表示。如图11－13所示。

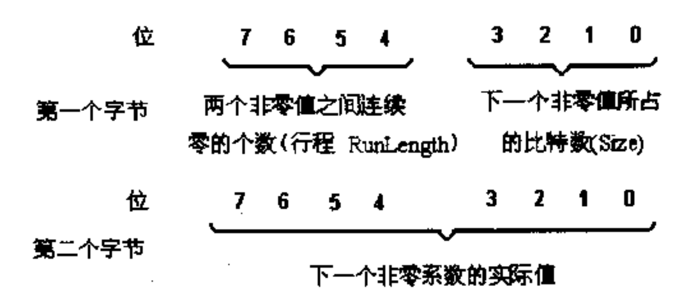


图11－13　行程编码

按照上面的方法，可以得到了DC码字和AC行程码字。为了进一步提高压缩比，需要对其再进行熵编码，这里选用Huffman编码。编码分成两步：

（1）熵编码的中间格式表示：

对于AC系数，有两个符号。符号1：行程和尺寸，即上面的（RunLength，Size）。（0，0）和（15，0）是两个比较特殊的情况。（0，0）表示块结束标志（EOB），（15，0）表示ZRL，

当行程长度超过 15 时，用增加 ZRL 的个数来解决，所以最多有三个 ZRL（3*16+15=63）。符号 2 为幅度值（Amplitude）。

对于 DC 系数，也有两个符号：符号 1 为尺寸（Size）；符号 2 为幅度值（Amplitude）。

（2） 熵编码：

对于 AC 系数，符号 1 和符号 2 分别进行编码。零行程长度超过 15 个时，有一个符（15，0），块结束时只有一个符号（0，0）。

对符号 1 进行 Hufffman 编码（亮度，色差的 Huffman 码表不同）。对符号 2：进行变长整数 VLI 编码，举例来说：Size＝6 时，Amplitude 的范围是-63～-32，以及 32～63，对绝对值相同，符号相反的码字之间为反码关系。所以 AC 系数为 32 的码字为 100000，33 的码字为 100001，-32 的码字为 011111，-33 的码字为 011110。符号 2 的码字紧接于符号 1 的码字之后。

对于 DC 系数，Y 和 C_bC_r 的 Huffman 码表也不同。下面举例来说明上述的编码过程。

下面为 8*8 的亮度（Y）图像子块经过量化后的系数：

15	0	-1	0	0	0	0	0
-2	-1	0	0	0	0	0	0
-1	-1	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

可见量化后只有左上角的几个点（低频分量）不为零，这样采用行程编码就很有效。

第一步，熵编码的中间格式表示：先看 DC 系数。假设前一个 8*8 子块 DC 系数的量化值为 12，则本块 DC 系数与它的差为 3，根据下表

Size	Amplitude
0	0
1	-1,1
2	-3,-2,2,3
3	-7~-4，4~7
4	-15~-8，8~15
5	-31~-16，16~31
6	-63~-32，32~63
7	-127~-64，64~127
8	-255~-128，128~255
9	-511~-256，256~511
10	-1023~-512，512~1023
11	-2047~-1024，1024~2047

查表得 Size=2，Amplitude=3，所以 DC 中间格式为（2）（3），AC 系数接着被编码，经过 Zig-Zag 扫描后，遇到的第一个非零系数为-2，其中遇到零的个数为 1（即 RunLength）。

根据下面这张 AC 系数表

Size	Amplitude
1	-1,1
2	-3,-2,2,3
3	-7~-4，4~7
4	-15~-8，8~15
5	-31~-16，16~31
6	-63~-32，32~63
7	-127~-64，64~127
8	-255~-128，128~255
9	-511~-256，256~511
10	-1023~512，512~1023

查表得 Size=2。所以 RunLength=1，Size=2，Amplitude=3，所以 AC 中间格式为（1，2）（-2）。

其余的点类似，可以求得这个 8*8 子块熵编码的中间格式为：

（DC）（2）（3），（1，2）（-2），（0，1）（-1），（0，1）（-1），（0，1）（-1），（2，1）（-1），（EOB）（0，0）

第二步，熵编码：

对于（2）（3）：2 查 DC 亮度 Huffman 表得到 11，3 经过 VLI 编码为 011

对于（1，2）（-2）：（1，2）查 AC 亮度 Huffman 表得到 11011，-2 是 2 的反码，为 01

对于（0，1）（-1）：（0，1）查 AC 亮度 Huffman 表得到 00，-1 是 1 的反码，为 0

……

最后这一 8*8 子块亮度信息压缩后的数据流为 11011，1101101，000，000，000，111000，1010。总共 31 比特，其压缩比是 64*8/31=16.5，大约每个像素用半个比特。

可见，压缩比和图像质量是呈反比的，以下是压缩效率与图像质量之间的大致关系，可以根据需要，选择合适的压缩比。

表 11－7　　压缩比和图像质量

压缩效率（单位：bits/pixel）	图像质量
0.25～0.50	中~好，可满足某些应用
0.50～0.75	好~很好，满足多数应用
0.75～1.5	极好，满足大多数应用
1.5～2.0	与原始图像几乎一样

以上我们介绍了 JPEG 压缩的原理，其中 DC 系数使用了预测编码 DPCM，AC 系数使用了变换编码 DCT，二者都使用了熵编码 Huffman，几乎所有传统的压缩方法在这里都用到了。这几种方法的结合正是产生 JPEG 高压缩比的原因。

11.5.2 JPEG 的文件格式

JPEG 文件大体上可以分成以下两个部分：标记码（Tag）和压缩数据。先介绍标记码部

分。

标记码部分给出了JPEG图像的所有信息(有点类似于BMP中的头信息,但要复杂的多),如图像的宽、高、Huffman 表、量化表等。标记码有很多,但绝大多数的 JPEG 文件只包含几种。标记码的结构为:

SOI

DQT

DRI

SOF0

DHT

SOS

…

EOI

每个标记码都由两个字节组成,其中高字节固定为0xFF。每个标记码之前可以填上个数不限的填充字节0xFF。

下面介绍一些常用的标记码的结构及其含义。

1. SOI(Start of Image)

标记结构　字节数

0xFF　1

0xD8　1

任何JEPG文件都以该标记开头,因此可以将该标记作为判断一个图像文件是否为JPEG格式文件的依据。

2. APP0(Application)

APP0是JPEG保留给应用程序(Application)使用的标记码,而JFIF(JEPG File Interchange Format,由C-Cube Microsystems公司制定的一种JEPG文件交换格式)将文件的相关信息定义在此标记中。

APP0标记码的结构如表11-8所示。

表11-8　APP0标记码的结构

标记结构	字节数	含义
0xFF	1	
0xE0	1	
Lp	2	APP0标记码长度,不包括前两个字节0xFF,0xE0
Identifier	5	JFIF识别码 0x4A,0x46,0x49,0x46,0x00
Version	2	JFIF版本号 可为0x0101或者0x0102
Units	1	单位,等于零时表示未指定,为1表示英寸,为2表示厘米
Xdensity	2	水平分辨率
Ydensity	2	竖直分辨率
Xthumbnail	1	水平点数

续表

标记结构	字节数	含义
Ythumbnail	1	竖直点数
RGB0	3	RGB 的值
RGB1	3	RGB 的值
…		
RGBn	3	RGB 的值，n=Xthumbnail×Ythumbnail

3. DQT（Define Quantization Table）

DQT 标记码的结构如表 11－9 所示。

表 11－9 DQT 标记码的结构

标记结构	字节数	意义
0xFF	1	
0xDB	1	
Lq	2	DQT 标记码长度，不包括前两个字节 0xFF，0xDB
(Pq,Tq)	1	高四位 Pq 为量化表的数据精确度，Pq=0 时，Q0~Qn 的值为 8 位，Pq=1 时，Qt 的值为 16 位，Tq 表示量化表的编号，为 0~3。在基本系统中，Pq=0，Tq=0~1，也就是说最多有两个量化表。
Q0	1 或 2	量化表的值，Pq=0 时，为一个字节，Pq=1 时，为两个字节
Q1	1 或 2	量化表的值，Pq=0 时，为一个字节，Pq=1 时，为两个字节
…		
Qn	1 或 2	量化表的值，Pq=0 时，为一个字节，Pq=1 时，为两个字节。其中 n 的值为 0～63，表示量化表中 64 个值（之字形排列）

4. DRI（Define Restart Interval）

此标记需要用到最小编码单元（MCU，Minimum Coding Unit）的概念。前面提到，Y 分量数据重要，UV 分量的数据相对不重要，所以可以只取 UV 的一部分，以增加压缩比。目前支持 JPEG 格式的软件通常提供两种取样方式 YUV411 和 YUV422，其含义是 YUV 三个分量的数据取样比例。举例来说：如果 Y 取四个数据单元，即水平取样因子 Hy 乘以垂直取样因子 Vy 的值为 4，而 U 和 V 各取一个数据单元，即 $Hu\times Vu=1,Hv\times Vv=1$，那么这种部分取样就称为 YUV411。如图 11－14 所示。

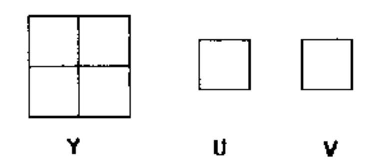


图 11－14 YUV411 的示意图

因为 YUV411 有 50%的压缩比（原来有 12 个数据单元，现在有 6 个数据单元），YUV422 有 33%的压缩比（原来有 12 个数据单元，现在有 8 个数据单元）。那么若采用 YUV911，

YUV1611 压缩比可以更高。但考虑到图像质量的因素，JPEG 标准也规定了最小编码单元 MCU，要求 $Hy \times Vy + Hu \times Vu + Hv \times Vv \leq 10$。

MCU 中块的排列方式与 H，V 的值有密切关系。如图 11－15 至图 11－17 所示。

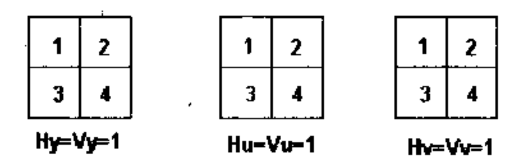


图 11－15 YUV111 的排列顺序

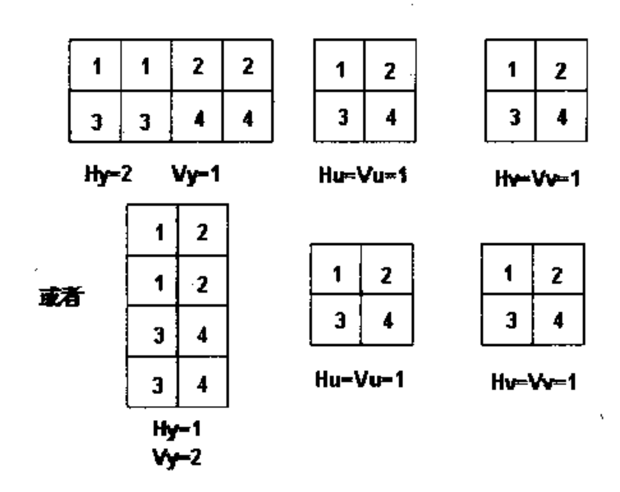


图 11－16 YUV211 的排列顺序

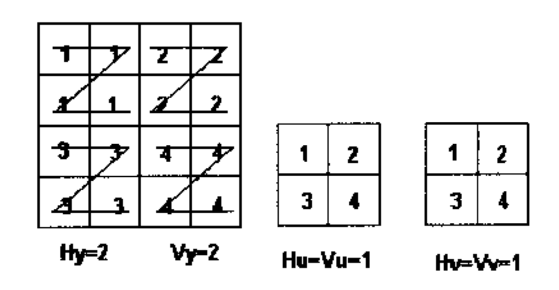


图 11－17 YUV411 的排列顺序

DRI 标记码的结构如表 11－10 所示。

表 11－10 DRI 标记码的结构

标记结构	字节数	意义
0xFF	1	
0xDD	1	

续表

标记结构	字节数	意义
Lr	2	DRI 标记码长度，不包括前两个字节 0xFF，0xDD
(Pq,Tq)	1	高四位 Pq 为量化表的数据精确度，Pq=0 时，Q0~Qn 的值为 8 位，Pq=1 时，Qt 的值为 16 位，Tq 表示量化表的编号，为 0~3。在基本系统中，Pq=0，Tq=0~1，即最多有两个量化表。
Q0	1 或 2	量化表的值，Pq=0 时，为一个字节，Pq=1 时，为两个字节
Q1	1 或 2	量化表的值，Pq=0 时，为一个字节，Pq=1 时，为两个字节
…		
Ri	2	重入间隔的 MCU 个数，Ri 必须是一 MCU 行中 MCU 个数的整数，最后一个零头不一定恰好是 Ri 个 MCU。每个重入间隔各自独立编码。

5. SOF（Start of Frame）

表 11－11　　DRI 标记码的结构

标记结构	字节数	意义
0xFF	1	
0xC0	1	
Lf	2	SOF 标记码长度，不包括前两个字节 0xFF，0xC0
P	1	基本系统中，为 0x08
Y	2	图像高度
X	2	图像宽度
Nf	1	Frame 中的成分个数，一般为 1 或 3，1 代表灰度图，3 代表真彩图
C1	1	成分编号 1
(H1,V1)	1	第一个水平和垂直采样因子
Tq1	1	该量化表编号
C2	1	成分编号 2
(H2,V2)	1	第二个水平和垂直采样因子
Tq2	1	该量化表编号
…		
Cn	1	成分编号 n
(Hn,Vn)	1	第 n 个水平和垂直采样因子
Tqn	1	该量化表编号

6. DHT（Define Huffman Table）

表 11－12　　DRI 标记码的结构

标记结构	字节数	意义
0xFF	1	
0xC4	1	

续表

标记结构	字节数	意义
Lh	2	DHT 标记码长度，不包括前两个字节 0xFF，0xC4
(Tc,Th)	1	
L1	1	
L2	1	
…		
L16	1	
V1	1	
V2	1	
…		
Vt	1	

Tc 为高 4 位，Th 为低 4 位。在基本系统中，Tc 为 0 或 1，为 0 时，指 DC 所用的 Huffman 表，为 1 时，指 AC 所用的 Huffman 表。Th 表示 Huffman 表的编号，在基本系统中，其值为 0 或 1。

所以，在基本系统中，最多有 4 个 Huffman 表，如下所示：

Tc　Th　Huffman 表编号（2*Tc+Th）

0　0　0

0　1　1

1　0　2

1　1　3

Ln 表示每个 n 比特的 Huffman 码字的个数，n=1~16

Vt 表示每个 Huffman 码字所对应的值，也就是我们前面所讲的符号 1，对 DC 来说该值为(Size)，对 AC 来说该值为（RunLength，Size）。

t=L1+L2+…L16

7. SOS(Start of Scan)

表 11－13　　DRI 标记码的结构

标记结构	字节数	意义
0xFF	1	
0xDA	1	
Ls	2	SOS 标记码长度，不包括前两个字节 0xFF，0xDA
Ns	1	
Cs1	1	
(Td1, Ta1)	1	
Cs2	1	
(Td2, Ta2)	1	
…		

续表

标记结构	字节数	意义
CsNs	1	
(TdNs, TaNs)	1	
Ss	1	
Se	1	
(Ah, Al)	1	

Ns 为 Scan 中成分的个数，在基本系统中，Ns=Nf（Frame 中成分个数）。CSNs 为在 Scan 中成分的编号。TdNs 为高 4 位，TaNs 为低 4 位，分别表示 DC 和 AC 编码表的编号。在基本系统中 Ss=0，Se=63，Ah=0，Al=0。

8. EOI(End of Image) 结束标志

标记结构	字节数	意义
0xFF	1	
0xD9	1	

11.5.3 编程实现 JPEG 文件的读写

编程实现 JPEG 文件的读写比较复杂，但是因特网上有很多现成的源代码可以直接使用。所以限于篇幅这里就不给出源程序了。